A practical guide to writing a radiative transfer code

S. Korkin[a,b], A. M. Sayer[a,b], A. Ibrahim[b] and A. Lyapustin[b]

[a] Universities Space Research Association GESTAR, Columbia, MD, USA.
[b] NASA Goddard Space Flight Center, Greenbelt, MD, USA.

"You can't trust code that you did not totally create yourself. "

--Ken Thompson [1]

Contents

Abstract

Using our decades-long experience in radiative transfer (RT) code development for Earth science, we endeavor to reduce the knowledge gap of bringing RT from theory to code quickly. Despite numerous classic and recent literature, it is still hard to develop an RT code from scratch within a few weeks. It is equally hard to understand, not to mention modify, an existing "monster" RT code, for which the developer is either located remotely or has retired. Following the format of "Numerical Recipes" by Press et al., we collocate in this paper small pieces of necessary theory with corresponding small pieces of RT code. These are arranged in an order that is natural for code development, which is often opposite of the natural order for laying out the theoretical basis. We focus on the transfer of unpolarized monochromatic solar radiation in a plane-parallel atmosphere over a reflecting surface. Both the surface and the atmosphere are homogeneous (uniform) at all directions. The multiple scattering is numerically solved using the deterministic method of Gauss-Seidel iterations. Except for the presented Python-Numba open-source RT code `gsit`, the paper does not report any new scientific results, but rather serves as an academic demonstration. If development time is an issue or the reader is familiar with basic concepts of RT theory, we recommend proceeding directly to Sec.3 "RT code development".

Program summary

Program title: gsit (pronounced "jeezit")

CPC Library link to program files: (to be added by Technical Editor)

Developer's repository link: https://github.com/korkins/gsit

Code Ocean capsule: (to be added by Technical Editor)

License: MIT

Programming language: Python 3

Nature of problem: We present a tutorial in Python code for deterministic (non-stochastic) numerical simulation of multiple scattering of monochromatic solar light in a plane-parallel Earth atmosphere bounded from below by a reflecting surface. The problem is solved in a simplified form (i.e., uniform atmosphere, no polarization, uniform surface reflectance, etc.) to better explain numerical features, rather than physics, of propagation of light in the atmosphere.

Solution method: The method of Gauss-Seidel iterations. It relies on the Fourier decomposition of the Radiative Transfer Equation over azimuth, Gauss quadrature for numerical integration over the zenith and iterative process for integration over height (optical depth) with analytical (hence known) single scattering approximation being the starting point. The method is relatively simple to code and does not require any external libraries.

Keywords

Multiple light scattering, Earth science, software development, Radiative transfer equation, Gauss-Seidel iterations.

1. Introduction

Table 1: List and definition of acronyms used in this paper

Acronym	Definition	Type
6S	Second Simulation of the Satellite Signal in the Solar Spectrum	RT code
(A)ATSR	(Advanced) Along-Track Scanning Radiometer	Space-borne instrument
AD	Adding-Doubling	RT technique
AERONET	AErosol RObotic NETwork	Sun photometer network
AI	Artificial Intelligence	Branch of computer science
ARM	Atmospheric Radiation Measurement	US climate research facility
ARTS	Atmospheric Radiative Transfer Software	RT code
BOA	Bottom Of Atmosphere	Physical term
BRDF	Bidirectional Reflectance Distribution Function	Physical term
CPU	Central Processing Unit	Computational term
DISORT	DIScrete Ordinates Radiative Transfer	RT code
DO	Discrete Ordinates	RT technique
FORTRAN	FORmula TRANslation	Programming language
GPU	Graphics Processing Unit	Computational term
GS	Gauss-Seidel	RT technique
GSIT	Gauss-Seidel Iterations	RT code (from this work)

IE	Invariant Embedding	RT technique
IPOL	Intensity and POLarization	RT code
libRadTran	Library for Radiative Transfer	RT code
LUT	Look-Up Table	Computational term
MAIAC	Multi-Angle Implementation of Atmospheric Correction	Retrieval algorithm
MC	Monte-Carlo	Numerical technique relying on random (stochastic) sampling
MO	Matrix Operator	RT technique
MODTRAN	MODerate resolution atmospheric TRANsmission	RT code
MPI	Message Passing Interface	Computational term
OpenMP	Open Multi-Processing	Computational term
RT	Radiative Transfer	Physical term
RTE	Radiative Transfer Equation	Physical term
SASKTRAN	Not explicit; derives from Saskatoon	RT code
SBDART	Santa Barbara DISORT Atmospheric Radiative Transfer	RT code
SCIATRAN	Not explicit; derives from SCanning Imaging Absorption SpectroMeter for Atmospheric CHartographY (SCIAMACHY)	RT code
SH	Spherical Harmonics	RT technique
SHARM (3D)	Spherical HARMonics (with 3D effects)	RT code
SKIRT	Stellar Kinematics Including Radiative Transfer	RT code
SMART-G	Speed-Up Monte-Carlo Advanced Radiative Transfer code with GPU	RT code
SO	Successive Orders	RT technique
SORD	Successive ORDers	RT code
TOA	Top Of Atmosphere	Physical term
TOMRAD	Total Ozone Mapping Spectrometer (TOMS) Radiative Transfer Model	RT code
(V)LIDORT	(Vector) Linearized Discrete Ordinate Radiative Transfer	RT code
XRTM	X Radiative Transfer Model	RT code

Radiative transfer (RT) codes are scientific software that numerically simulate the propagation of electromagnetic radiation through a medium. RT simulations are used in various disciplines including astrophysics, planetary and Earth science, and remote sensing. RT codes are a fundamental component in remote sensing retrieval algorithms of geophysical parameters from space- and air-borne

observations, as well as ground-based measurements of the Earth system. RT codes need to satisfy three requirements: 1) accurately model the physical phenomena of radiative transfer, 2) be numerically stable, and 3) be sufficiently fast. This is an acronym-heavy field: for convenience, all acronyms used in this paper are defined in Table 1.

The transfer of radiation is governed by a fundamental equation that describes the variation of light intensity in a medium characterized by its scattering, absorption, and emission. In the Earth system, RT requires a set of boundary conditions such as the illumination by extraterrestrial light and surface boundary, like reflectance from land or wind-ruffled ocean. Additionally, best software development practices such as version control, unit testing, readability, and maintainability should be followed when developing an RT code. Proper documentation is necessary for a non-developer user. In this paper we focus on the software side of RT code development.

The term "RT code" can refer to programs with very different levels of complexity, dependent on what exactly is included. Here, by "RT code" we mean only the part responsible for the numerical solution of the RT equation (RTE). This part is often also called the "RT solver". It deals only with the inherent parameters of the RTE, and accuracy parameters of the numerical method chosen to solve the RTE. For the RT solver to be used in applications, it must be equipped with plug-ins that compute RTE parameters (e.g. single-scattering properties) for relevant quantities used in the application. The RT solver in combination with auxiliary software compose a fully functional RT code (often called RT model or library), for example, Fig.1 of libRadtran [2] or the MODTRAN[1] and Ocean Optics Web Book[2] websites. RT libraries often combine different RTE solvers, lots of auxiliary software to compute parameters of the RT models, and instrument parameters, such as satellite spectral response functions, and built-in interfaces for interaction with other models and databases [2–4].

In what follows, we use the words "*radiation*" and "*light*" as synonyms. To be specific, we will be talking about Earth science, and within the solar spectral domain. However, astrophysics, planetary science, nuclear physics, and computer graphics, as well as Earth science applications in other spectral ranges, deal with similar problems. We therefore hope our paper will be useful for readers beyond our domain.

The development of RT codes began more than half a century ago as access to computing power for scientific purposes become more widespread. Some codes, developed by experts in their fields, have remained in use for decades. For example, arguably the oldest RT code TOMRAD, developed in the 1960s [5], is still actively used to study atmospheric ozone [6,7], nitrogen [8,9], and volcanic sulfur dioxide [9]. DISORT, the most cited RT code in Earth science[3], was developed in the 1980s [10]. It is still supported [11] and used worldwide [12]. Long lasting use of codes assures numerical accuracy and stability, and most importantly broad validation. However, as computer and Earth science move forward, limitations of heritage codes previously considered sufficient become more pressing, such as missing physical effects (e.g., new atmospheric absorption data), or advancements in high performance computing (e.g., parallel processing, GPUs, and accelerated software) since the time the code was originally developed. The latter is severely underutilized in almost all widely-used RT codes, SMART-G being an exception [13].

[1] http://modtran.spectral.com/modtran_about
[2] https://www.oceanopticsbook.info/view/radiative-transfer-theory/radiative-transfer-equations
[3] https://www.osapublishing.org/ao/journal/ao/anniversary/50mostcited.cfm

Note that the creation of a mature RT code suitable for real world applications (e.g. retrievals from remote sensing) takes years of collaborative efforts. libRadtran "*was initiated about 20 years ago and is still under continuous development*" [2]. ARTS has had a similar lifespan[4]: 19 years as of 2021. Other fields such as astrophysics, show similar situations. The Monte Carlo (MC, a stochastic method) RT code SKIRT[5] was first published in 2003 [14]. After years of usage and development, in 2015, it was released with a user- and application- friendly interface [15,16], MPI support in 2017 [17], and most recently released as v.9 in 2020 [18] after refactoring.

Modifying current RT codes to include more advanced capabilities is a difficult and slow process. Most importantly, legacy RT code developers are retiring and increasingly unavailable, rendering these legacy codes as "black boxes". This is typically due to insufficient documentation, lack of version control, and obscure coding practice (whether due to personal style or old language conventions), combined with the sophisticated mathematical formalism behind RT codes. This is a common problem in scientific software development [19,20]. The current situation with RT code development is summarized in [21]: "*Although this may often not be the case, we will assume in what follows that direct modeling is performed adequately …* ".

Over time, the need for detailed understanding of RT code structure to provide efficient support and, if needed, complete refactoring, has increased. However, the existing literature does not offer a quick solution. Numerous books and papers provide the theoretical background behind numerical methods in RT. The most notable is "Radiative transfer in scattering and absorbing atmospheres: standard computational procedures"[22], written by a group of world-leading RT code developers. The book explains the idea and theoretical background for most, if not all, methods for RTE solution; advantages and disadvantages for each; and provides benchmark results and a broad list of references available at that time. Software aspects of RT code and mathematical formalism for polarization of light are not considered, though the latter problem is discussed in [23] and [24]. Ideas behind different techniques for numerical simulation of multiple light scattering are discussed in [23,25,26]. Classic volumes by founders of the RT theory [27,28] and numerical methods [29] focus on physics and intensive math however provide little guidance on RT code development.

Expanding beyond the mathematical foundation of RT theory, manuals or user guides available for some RT solvers may provide an alternative approach to better understand the RT code development process, however they prove to be insufficient. For example, Section 9, "Accurate Numerical Solutions of Prototype Problems" in [30] is an updated DISORT manual. This manual focuses on the theoretical background for the code and explains input and output, but not the code structure and implementation. A detailed description of RT package SCIATRAN is available in [31]. In addition to acting as a manual, this review explains the theoretical background for each SCIATRAN's feature: multiple scattering, absorption, surface models, effect of sphericity of the planetary atmospheres, built-in spectroscopy, etc. The RT codes (V)LIDORT [32] and 6S [33], RT library libRadtran [2] among others come with extensive and user-friendly manuals; ARTS[6] and SASKTRAN[7] and 6S[8] are described on the web.

[4] https://en.wikipedia.org/wiki/ARTS_(radiative_transfer_code)
[5] https://skirt.ugent.be/
[6] https://www.radiativetransfer.org/
[7] https://arg.usask.ca/projects/sasktran/
[8] http://6s.ltdri.org/

Unfortunately, without in-depth study of thousands of lines in corresponding codes, these manuals can't increase the reader's knowledge from user to developer level alone. This level of understanding is required when someone other than the developer faces the task of upgrading and/or debugging a complex code, or even worse, the problem of developing of an in-house code from scratch.

Pieces of information on RT code development are rare and scattered across numerous literature sources. *Hansen & Pollack* [34] report "shortcuts" for multiple scattering computations and estimate the gain in runtime for each. *Hansen & Travis* [23] analyze several RT techniques and derive practical suggestions for their implementation. *Van de Hulst* [35] indicates a flowchart on how to simulate successive orders of scattering. Sections 2 and 5 in [36] discuss requirements for RT code and optimization of computational parameters, respectively. *Efremenko* et al. [37] report on implementation of multi-core CPU and GPU tools in RT codes, pseudo-codes for RT solver as used in retrieval algorithm, the use of parallel computations in RT models, and GPU architecture – a rare example where the software side of the RT code development dominates over theoretical background. Full code of Fortran subroutines to be used in the methods of successive orders in atmospheres with arbitrary profiles of optical parameters are reported in [38].

Using our decades-long experience in RT code development [39–43] and application in Earth science [44,45,54,46–53], in this paper we endeavor to reduce the knowledge gap of bringing the RTE from theory to code quickly. In Section 2 we describe the theoretical steps for RTE solution and concisely outline a few widely-used numerical techniques. The section is intended to introduce our reader to the problem and provide a general overview of the RTE solution algorithm. Then in Section 3 we focus on the theory (i.e., equations) and corresponding Python function for each particular step. For faster development, the reader can start directly from Section 3. We present the code co-located along with corresponding equations. The format is inspired by the approach used in "Numerical recipes" book series, starting from Pascal [55] and ending with object-oriented C++[56]. We chose Python as a popular modern language, although it may not be as computationally efficient as some others. By doing that we follow the "*make it right – then make it fast*" philosophy, placing clarity as our primary goal. Python is frequently used in atmospheric science in general [57] and in RT code development in particular (e.g. refer to websites of libRadtran[9], SASKTRAN[10], or XRTM[11]), but mostly as an interface or wrapper to an RT solver traditionally developed in Fortran and/or C [58]. We have, however, accelerated our Python code using a high-performance Python compiler, Numba[12]. As a result, Section 3 offers a fully working RT code based on the method of Gauss-Seidel iterations with 2 benchmarks (in Appendices), as well as data and suggestions for unit testing. Section 4 discuss some ideas for further development of the code (vertical profiles, surface BRDF), numerical efficiency and structure. We conclude the paper with a Summary.

Several RT tools have been developed with teaching in mind. For example , SBDART [59] has had an online interface since the 1990s; libRadtran was used to model clouds during ARM training [60],

[9] www.libradtran.org
[10] https://arg.usask.ca/projects/sasktran/
[11] https://reef.atmos.colostate.edu/~gregm/xrtm/
[12] https://numba.pydata.org/

HydroLight, an RT model for light scattering in ocean, comes with an extensive "Ocean Optics" web book[13], whose section "Radiative Transfer Theory" is directly related to this paper.

We approach this paper as an academic demonstration, and don't expect our reader to have any knowledge of numerical methods or coding experience other than those taught during undergraduate degrees. That necessitates a simplification to RT theory by not accounting for some physical principles, such as assuming the atmosphere to be plane-parallel (1-dimensional), non-emitting, and homogeneous (i.e. one optical layer) irradiated from above by a monochromatic infinitely wide solar beam at an arbitrary angle. We neglect polarization of light (scalar RT). The atmosphere is bounded from below by a surface that reflects light evenly over angles (i.e. Lambertian reflection).

The dependence of atmospheric optical properties on height (profile), polarization of light, wave-covered ocean, landscape shadows, and many other effects are important for atmospheric, oceanic, and terrestrial applications but secondary in terms of goals of this paper. A "monster code" [61] that can do all these things is never all-powerful, and is always hard to learn and debug.

Further in the paper we deal with deterministic methods to solve the RTE. These methods rely on explicit analytical evaluations, such as decomposition in series (e.g., Fourier), linear algebra, numerical and analytical integration, and other formalism. Analytical approaches are methods of choice for cases with significant symmetry, such as 1D RT problem (horizontally homogeneous atmosphere). Given the fairly low spatial resolution of many Earth science instruments (of order 1-10 km), the 1D approximation has been widely used for applications. For retrievals, the deterministic methods yield analytical linearization (computation of derivatives) [32,62]. For atmospheric correction, some deterministic methods allow one to "split" atmosphere and surface signals. The latter is significant due to spatial and temporal scales of variability: slow spatial and rapid temporal variations of atmosphere as opposed to surface, which may vary significantly within a few meters, while staying unchanged for weeks during a season. Therefore, it is efficient to precompute and store in a LUT the scattering and absorption properties of the atmosphere itself and apply those for different top (Sun moving during the day) and bottom (rapid spatial variations of surface) boundary conditions. An example of this is the combination of 3D surface parameters with 1D atmosphere using the Green function formalism [43,63]. Another example is SHDOM[14]: combination of analytical methods, SH and DO, to account for atmospheric 3D effects such as cloud shape [64,65]. The plane-parallel approach is often generalized for 3D cases using independent pixel approximation (unique 1D RT problem is solved within each pixel of satellite image, horizontal energy flow is either ignored or approximated). Complex shapes of real water clouds or dust plumes can be replaced with idealized "brick" shapes for efficient deterministic simulations; the accuracy of this approximation can then be estimated by comparing with MC simulations.

As opposed to deterministic approaches, MC considers propagation of light along the path as a random (stochastic) process. As a result, output from MC RT codes is an expectation (mean value) based on many simulated photons, with an estimated standard deviation as an uncertainty bound. The statistics of this random sampling, of course, depend uniquely on the optical properties of the scattering media, light source, and reflecting surface. The method of MC converges as the number of the random processes (light rays, or photons) increases. However, the convergence rate is inversely proportional to

[13] https://oceanopticsbook.info/
[14] https://nit.coloradolinux.com/shdom.html

the square root of the number of runs. Hence decreasing the random error by a factor of 10 requires running the MC code 100 times more. Because of this, MC codes are considered slow compared to analytical methods for problems that possess symmetry, such as plane-parallel atmosphere with no horizontal variations of optical properties, illuminated by an infinitely wide unidirectional beam. But MC RT codes becomes good (and sometimes the only feasible) option when the RT problem becomes less symmetric, e.g., radiative transfer close to edge of clouds with a sophisticated shape. Note that consideration of the polarization of light reduces symmetry of the problem even if the atmosphere and surface remain optically uniform. Polarization significantly complicates mathematical apparatus for deterministic polarized RT codes. That reason alone may be sufficient to go with faster development of the 1D MC RT code and sacrifice runtime.

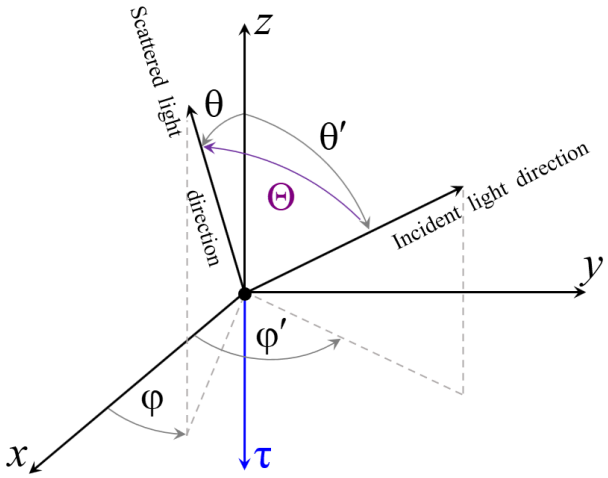
In what follows we do not discuss MC codes, but refer our reader to [66]. That paper discusses accuracy of several state-of-the-art MC RT codes (in addition to non-MC) widely used by atmospheric community. Beyond the scope of this paper we also leave the use of the artificial intelligence (AI) approach to solve the RTE [67–69] and the role of the RT solvers in training of the AI-based retrieval algorithms [70]. Now we proceed to a general description of the steps necessary to solve the RTE using deterministic (analytical) approach for a plane-parallel horizontally homogeneous atmosphere over homogeneous surface, illuminated by an infinitely wide ideally collimated source of light – the Sun.

2. Steps to solve the RTE

2.1. Definition of the RTE

The RTE describes the dependency of intensity of radiation, I, on location and direction of propagation in a medium. The intensity is expressed in units of power (in Watts, W) per unit area (in meters squared, m^2) oriented perpendicular to the direction of propagation of light, per unit solid angle (in steradians, sr), per unit band (in atmospheric optics: nanometers or microns in the wavelength domain, but temporal and spatial frequencies are also used in atmospheric spectroscopy). In the plane-parallel atmosphere, the location of interest is characterized by the vertical coordinate z (i.e. height) only.

A beam of light traversing through the atmosphere will be absorbed and/or scattered (i.e. redistribute the energy in different directions). To describe the absorption and scattering properties of the atmosphere we use a dimensionless optical depth τ measured from the top of the atmosphere (TOA) to a point of interest along the coordinate z. The optical depth combines the inherent optical property of the medium (extinction efficiency) with the physical distance the light travels. Zenith θ and azimuth φ angles specify the direction (**Fig.1**: left). The optical length of a segment AB (**Fig.1**: right) in the atmosphere is τ/μ, $\mu=-\cos(\theta)$, where the minus sign is a convention caused by opposite directions of the z and τ axes. Solving the RTE means the ability to evaluate $I(\tau, \mu, \varphi)$ at each value of the 3 coordinate system for the given optical properties of the atmosphere with two boundary conditions: the intensity of the incoming light at TOA, and reflection at the bottom of atmosphere (BOA).



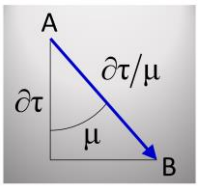


Fig.1 Left: Zenith θ and azimuth φ angles in Cartesian coordinate system. Forward scattering corresponds to zero azimuth (e.g., as if a light sensor pointing in the plane of the sun). Top: Optical path between points A and B.

In this paper, we adhere to a common convention in RT theory and numerical simulations: exact "forward" and "backward" scattering correspond to φ = 0° and φ = 180°, respectively (principal plane of the Sun). Hereinafter, we refer to the term "scattering" in a physical sense, i.e. relative to the direction of propagation of the light beam just before the moment of scattering – see **Fig.2**. "Forward scattering" refers to all directions with the scattering angle from 0 to 90 degrees (hemisphere), the rest is "backscattering". Therefore, the aureole is in the forward scattering hemisphere (almost precise forward scattering), when the observer is facing the Sun, while rainbow and glory are in the back (the latter is at almost precise backscattering). Note, forward (backward) scattering directions may or may not correspond to φ = 0° (180°) as **Fig.2** shows.

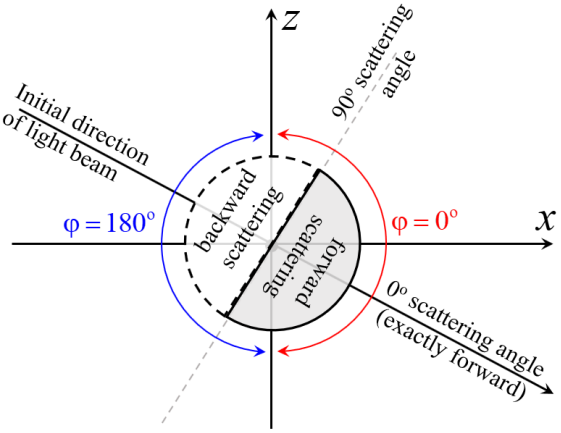


Fig.2: Definition of forward and backward scattering (hemispheres). Note, either may contain directions with relative azimuth φ=0° or 180°. The definition is similar for the surface reflection, except for $z > 0$.

The situation is similar for surface reflection, except the z-component of the vector that represents the light beam changes sign (light goes upward, instead of downward). According to the convention adopted in this paper, mirror reflection is at relative azimuth φ = 0° precisely (but not necessarily in the forward hemisphere), glint belongs to φ < 90°, hotspot belongs to φ > 90°. In satellite remote sensing, the definition is often the opposite: glint is observed at azimuths exceeding 90°. This is sometimes known colloquially as the "Gordon convention", which we do not use in this paper. **Fig.3** shows the difference between the two conventions. We include the above discussion on this point because we have found it a common stumbling block when atmospheric scientists begin to learn to use RT codes.

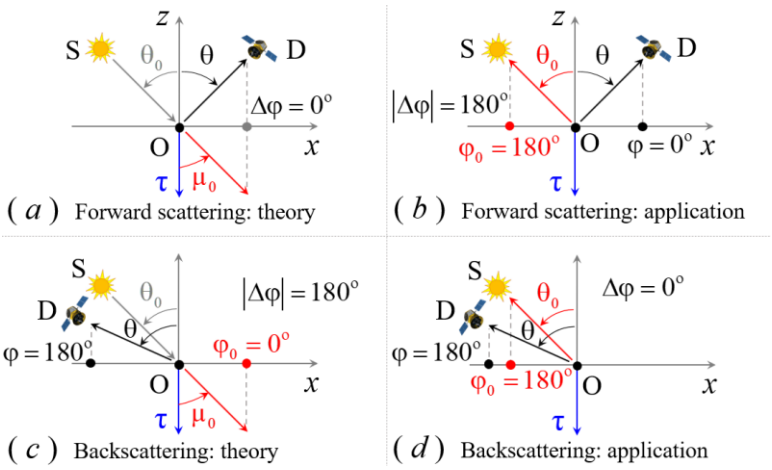


Fig.3 Relative azimuth convention for theoretical evaluations (this paper) - cases (a) and (c) vs. the one often used in applications – cases (b) and (d). Ocean glint is observed when the detector D faces the Sun, S. In this paper, we say the relative azimuth is zero (a), while the application convention says it is 180° (b). Same for the hotspot – cases (c) and (d). For us, the hotspot happens at relative azimuth 180 ° (c), while for applications it could be 0 ° (d).

In the scalar RTE, the optical properties of the atmosphere can be defined through three parameters: the optical thickness τ, single scattering albedo ω_0, and the phase function $p(\Theta)$. The single scattering albedo is the ratio of the scattering to the total extinction (i.e. absorption + scattering). It is the likelihood for light to survive a scattering event: ω_0=0 means all interactions result in absorption, ω_0=1 means all result in light scattering. The phase function $p(\Theta)$ is the scattering probability distribution function, dependent on the scattering angle Θ (**Fig.1**: left). The scattering angle is the angle between incident and scattered light directions. In a Cartesian system of coordinates, it can be expressed via zeniths and azimuths of the incident and scattered light. The surface at BOA is characterized by a bidirectional reflectance distribution function (BRDF), ρ. The BRDF acts like $p(\Theta)$, except the light gets reflected (light changes direction from "down" to "up"), not scattered (light changes direction arbitrary or not at all).

The change of the intensity of light as it travels in the atmosphere in the given direction is expressed by a derivative with respect to τ (i.e. the rate of change in intensity w.r.t. change in optical depth) (**Fig.1**: right) – left side in Eq.(1) below. Redirection of energy through absorption and scattering along the path of travel reduces the intensity of the traveling light (hence minus sign in the first term in the right side of Eq.(1)). Scattering accumulated (integrated) from all directions into the direction of propagation increases the intensity (second and third terms in the right side of Eq.(1)). This yields the RTE in integro- (over θ, φ) differential- (over τ) form:

$$\mu\frac{\partial I(\tau,\mu,\varphi)}{\partial\tau} = -I(\tau,\mu,\varphi) + \frac{\omega_0}{4\pi}p(\mu,\mu_0,\varphi,\varphi_0)I_0\exp\left(-\frac{\tau}{\mu_0}\right) + J(\tau,\mu,\varphi) \qquad (1)$$

where

$$J(\tau,\mu,\varphi) = \frac{\omega_0}{4\pi}\int_0^{2\pi}\int_{-1}^{1} p(\mu,\mu',\varphi,\varphi')I(\tau,\mu',\varphi')d\mu'd\varphi' \qquad (2)$$

is the scattering integral. The second term in the right side of Eq.(1) describes first scattering of the direct solar beam. I_0 is the TOA spectral irradiance (W/m^2/nm). The solar geometry is defined by cosine of the solar zenith, μ_0=cos(θ_0), and azimuth φ_0. The latter is usually assumed zero, φ_0 = 0, in the solar (principal) plane.

Since Eq.(1) is linear, it is convenient to scale both sides by some factor. For example, if one scales both sides by $\frac{\pi}{\mu_0 I_0}$ (inversed units of intensity), the RTE will deal with unitless reflectance and transmittance (both may exceed 1). Often for simplicity the TOA flux is assumed unity (without explicit definition of its units), $I_0 = 1$ [40,71].

Setting $J(\tau,\mu,\varphi) = 0$, Eq.(2), yields the RTE for the single scattering approximation, $I_1(\tau,\mu,\varphi)$ (note subscript '1'). The single scattering approximation is often treated separately due to its analytical simplicity and importance for understanding of the physics of scattering (influence of different optical properties on solution), as well as for the acceleration of numerical simulations. In the case when the medium is non-scattering (i.e. ω_0=0), the second and third terms in the right part of Eq.(1) vanish, reducing the RTE to a first order differential equation whose solution is the Bouguer-Lambert-Beer (exponential) attenuation law.

2.2. Boundary conditions

As mentioned previously, the RTE solution requires two boundary conditions: incoming solar light at TOA and surface reflectance at BOA. The first is an initial condition, independent of the RTE solution. It is imposed on downward radiation at the top boundary, τ=0, in the forward hemisphere, $\mu = \mu_+ > 0$. Since there is no scattered light coming from space, the TOA boundary condition is simply zero.

The second depends on both the reflective properties of the surface and the RTE solution at the surface, which complicates the problem. Contrary to TOA, the bottom boundary condition is imposed on reflected radiation, $\mu = \mu_- < 0$ at the ground level τ_0. Two effects contribute to the radiation reflected at the bottom: reflection of the direct solar beam, and reflection of diffuse radiation coming down from the atmosphere (including multiple bouncing of radiation between the atmosphere and surface). For Lambertian surface reflection with surface albedo ρ, the bottom boundary condition is

$$I(\tau_0,\mu_-,\varphi) = \frac{\rho}{\pi}\mu_0 I_0 \exp\left(-\frac{\tau_0}{\mu_0}\right) + \frac{\rho}{\pi}\int_0^{2\pi}\int_0^1 \mu\, I(\tau_0,\mu,\varphi)\, d\mu\, d\varphi \qquad (3)$$

Comparing Eq.(3) with Eqs.(1) and (2), one can see that reflection of the direct solar beam is an equivalent of single scattering in the atmosphere, while multiple bouncing (i.e., diffused light) is like multiple scattering. The total intensity integrated over all directions yields irradiance (W/m2/nm), while scaling by inverse π (1/sr) gives intensity. The factor μ (μ_0 for the direct solar beam) in the integral accounts for the fact that the irradiance of oblique rays is smaller.

2.3. Methods for integrating the RTE

Eq.(1) is the RTE in integro-differential form. It is the starting point for methods based on replacing of the RTE with a system of differential equations such as Spherical Harmonics (SH) and Discrete Ordinates (DO). The methods of Gauss-Seidel (GS) and Successive Orders (SO) integrate over τ numerically. For that, they use the "formal solution": Eq.(1) explicitly integrated over τ. For numerical integration, the variable τ in the formal solution is discretized with some step Δτ. The τ-discretized RTE in integral form is:

$$I(\tau_i,\mu_\pm,\varphi)=I(\tau_{i\mp1},\mu_\pm,\varphi)\exp\left(\mp\frac{\Delta\tau}{\mu_\pm}\right)+I_1(\Delta\tau,\mu_\pm,\varphi)\pm\frac{1}{\mu_\pm}\int_{\Delta\tau}J(t,\mu_\pm,\varphi)\exp\left(\mp\frac{t}{\mu_\pm}\right)dt \qquad (4)$$

In Eq.(4), the terms have the following physical meaning (**Fig.4**): intensity at some level i (left hand side) is the sum of the intensity at the neighboring level above i-1 (for descending light, $\mu_+ > 0$) or below i+1 (for ascending light, $\mu_- < 0$) the current level, plus single scattering I_1 within the level $\Delta\tau$ (solar beam attenuated appropriately for embedded element layer $\Delta\tau$), plus multiple scattering within the element layer $\Delta\tau$ (last term in the right side with t indicating integration within $\Delta\tau$). Note the sign in exponential functions is opposite to that of μ and corresponds to the one in the subscript, i+1 (for ascending radiation) or i-1 (for descending radiation). The exponentials are responsible for the aforementioned Bouguer-Lambert-Beer attenuation. **Fig.4** indicates these components as (a), (b), and (c), respectively.

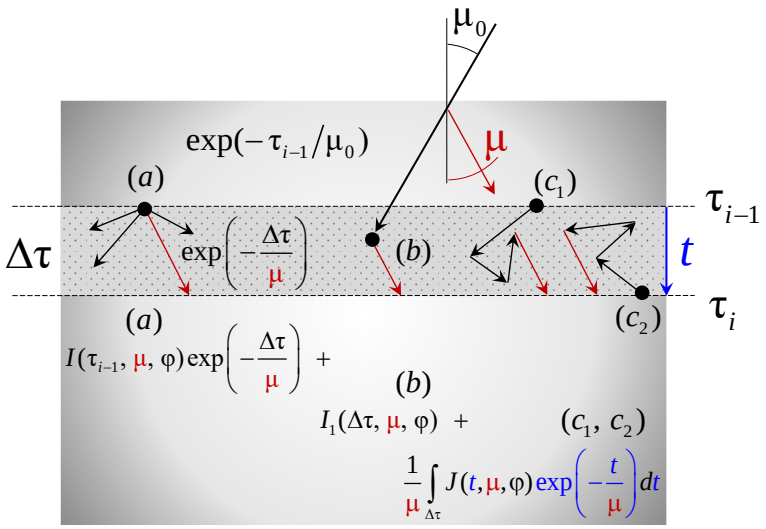


Fig.4: Illustration for Eq.(4), downward radiation: (*a*) direct contribution from the above layer, (*b*) single scattering contribution from the element layer $\Delta\tau$, (c_1) and (c_2) "re-scattering" of scattered radiation coming from above and below, respectively.

Eqs.(2) and (4) indicate that one deals with three integrals in this order: over azimuth φ, over cosine of the zenith angle μ, and over optical depth τ. Solving the RTE can be performed by numerical integration over the three variables in its simplest form as for example in [72]. However, in plane-parallel atmospheres, analytical integration over azimuth using Fourier expansion has become a standard technique. In the next three subsections, we will briefly describe the method of integration over each variable in the RTE.

2.3.1. Integration over the azimuth φ

Fourier expansion of the intensity

$$I(\tau,\mu,\varphi)=I^0(\tau,\mu)+2\sum_{m=1}^{M}I^m(\tau,\mu)\cos(m\varphi) \qquad (5)$$

and the phase function

$$p(\mu,\mu',\varphi,\varphi')=p^0(\mu,\mu')+2\sum_{m=1}^{M}p^m(\mu,\mu')\cos(m(\varphi-\varphi')) \qquad (6)$$

decouples the general RTE for $I(\tau, \mu, \varphi)$ into several independent RTEs for the Fourier *m*-moments, $I^m(\tau, \mu)$ with no azimuthal dependence in each (note the superscript *m*; $I_1^{m=1}$ would mean 1st Fourier harmonic for the first order of scattering). Because of the symmetry of the RTE solution $I(\tau, \mu, \varphi)$ with respect to the principal plane, **Fig.5**, and symmetry of the phase function with respect to the incident light direction, only $\cos(m\varphi)$ is used in Eqs.(5) and (6). It is only natural to convert the azimuth integration into a summation of Fourier series to reduce the integration steps. Aside from Monte Carlo methods, most modern RT codes for the plane-parallel atmosphere utilize azimuthal Fourier series decomposition.

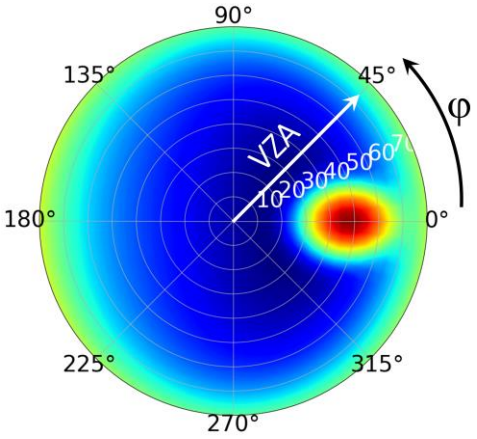


Fig.5: TOA intensity for a Rayleigh atmosphere over a wind-ruffled ocean [71] illustrate symmetry with respect to principal plane, φ=0°-180°. This explains the use of cosines in Eqs.(5) & (6). The highest (dark red) value corresponds to relative azimuths 0° ("mirror reflection" or glint, SZA=45°). The image is plotted in log-scale due to the peak of intensity at the glint exceeds the average intensity by an order of magnitude. VZA=0° corresponds to nadir observation (dark blue – weak signal).

Two steps are required for the Fourier expansion of the phase function. First, expand the phase function in Legendre polynomials as a function of scattering angle. Second, using the addition theorem for the Legendre polynomials, substitute the dependence on scattering angle with Legendre functions of incident and scattered zeniths and Fourier expansion over relative azimuth, which are native parameters of the RTE. These aspects are detailed in Sections 3.2 and 3.4, respectively.

2.3.2. Integration over the zenith cosine μ

There are various techniques to perform the integration over the cosine of the zenith angle, μ. Iterative techniques, like SO [5,73] and GS [72,74] , discretize the RTE and integrate over μ numerically to get the solution at the current iteration from the previous one, starting from single scattering (which is known analytically). Expansion of the phase function in Legendre polynomials favors the use of Gaussian quadrature integration over μ, for which the nodes are zeros of the Legendre polynomials. The method of Discrete Ordinates (DO) [10] also relies on Gaussian quadrature. However, intensities at the Gauss nodes are unknown and obtained from a system of differential (over τ) equations. The method of Spherical Harmonics (SH) [75,76] , in addition to Fourier series, expands the Fourier moments of intensity $I^m(\tau, \theta)$ in series over Legendre polynomials (see Section 3.2), as for the phase function, and uses orthogonal properties of the Legendre polynomials to get a system of equations for the Legendre *k*-moments of the Fourier *m*-moments of intensity, $I_k^m(\tau)$. Note there is no angular dependence in the *k*-moments. Once the *k*-moments are computed, the RTE solution becomes known at arbitrary set of the solar-view geometry, which is convenient for the computation of results at many view angles such as when generating lookup tables (LUTs) for use in satellite retrievals.

2.3.3. Integration over the optical thickness τ

Similar to integration over μ, integration over τ differs depending on the method. The methods of DO and SH, whose systems of differential equations are mathematically identical, integrate over τ analytically using eigenvalue decomposition of the system matrix [77]. Analytical integration yields high accuracy and independence of the codes' runtime on the optical thickness, an advantage when simulating for example thick clouds. In both techniques it is possible to split the atmosphere from the boundary conditions. However, the eigenvalue decomposition is time consuming and must be done for each optical layer, hence the use of optimized libraries is preferable. If runtime is an issue, these methods are advantageous for cases with a low (1-5) number of optical layers.

The iterative methods of SO and GS "know" the previous solution at some predetermined grid of τ-values and integrate over it numerically. This approach is simple (does not require external libraries) and efficient if the atmosphere is not very thick ($\tau \sim 1$). Numerical τ -integration is the method of choice for scenarios with complex vertical profiles (many optical layers) and/or smooth change of the solar-view geometry over height in the pseudo-spherical RT codes [78]. In the simplest (and often fastest) implementation, the SO and GS codes run anew for every solar zenith and surface condition simulated. Iterations start from the single scattering approximation, which is known.

A common technique used when simulating optically thick atmosphere is the Adding-Doubling (AD) method, which falls between numerical and analytical integration over τ. The method relies on the assumption that reflectance (R) and transmittance (T) properties from any (Gauss) direction to another (Gauss) direction is known. Hence, these properties of two neighboring layers combined can be computed from the individual properties – layers "adding". Starting from a very thin ($\tau \sim 2^{-10} \approx 0.001$) element layer, one computes T and R for a layer of arbitrary thickness. This method is very sensitive to the selection of the initial layer. However, in a uniform atmosphere two layers combined yield a layer of twice the optical thickness, which in turn is then combined with the neighboring layer of the same – doubled – thickness, and so on. The AD technique goes from TOA to BOA, includes all scattering orders, and relies on multiple matrix multiplications and inversion (the latter is to be replaced with the solution to the system of linear equations – hence optimized libraries are preferable). Invariant embedding (IE) and matrix operator (MO) methods use similar formalism to connect layers. Later in this paper we illustrate the main idea of the matrix operator technique (Section 3.6.2).

2.4. Choice of methods and general steps

The variety of techniques commonly employed to solve the RTE indicates that there is no one universal best method. The theoretical and numerical complexity of the solution grows as the problem becomes more complex for either atmosphere (e.g., many optical layers instead of one, 3D instead of 1D plane-parallel atmosphere), or boundary conditions (e.g., off-nadir Sun, azimuthally non-symmetric or inhomogeneous surface reflectance), or the radiation itself (e.g., polarization of light). It is for that reason the first step in solving the RTE is to pick a method that best fits the problem. The best code is not always the one that runs fastest: each problem also has its own accuracy requirements. One should also consider weighting between using a slower but easier to code, debug, and support method against a faster method that requires understanding of complicated mathematical background and needs optimized libraries (which may limit the code distribution due to software or licensing issues).

Numerical implementation of these techniques starts with coding the low-level functions first: Legendre polynomials, Gauss nodes and weights, and the single scattering approximation. The latter should be

done in two forms: first using single scattering without expansion in Fourier series over relative azimuth. It is sufficient to create this code only for boundaries where the RTE is to be solved (e.g., only at TOA). Second, one needs single scattering for a given Fourier order *m* at an arbitrary level in the atmosphere. This will be used as an initial guess in GS and SO to compute corresponding Fourier moments of the multiple scattering. Subtraction of the single scattering from the multiple scattering, once computed, yields the RTE solution for second and higher scattering orders expanded in Fourier series. At the final step, one accumulates the Fourier series, starting from the exact (not expanded) single scattering until sufficient accuracy is achieved. In the next section we present, step by step, the theoretical basis and show corresponding code for this algorithm using the method of GS iterations, which advantages are described in Section 3.6 “Numerical simulation of multiple scattering using Gauss-Seidel iteration”.

3. RT code development

In this section we will describe the mathematical representation of each of the RTE solution steps along with the associated Python implementation and routines.

3.1. Gaussian nodes and weights

Computation of Gaussian nodes, x_j, and weights, w_j, for numerical integration of a function $f(x)$

$$\int_{-1}^{1} f(x)dx \approx \sum_{j=1}^{2N} w_j f(x_j) \tag{7}$$

is very common [79]. Source code is available e.g. in Fortran [80], C [81], or directly from publicly-available RT codes (including `gsit` described in this paper). A built-in function is available e.g. in the Python packages `numpy.polynomial.legendre.leggauss`. **Fig.6** shows, without much discussion, the implementation of computation of the Gauss nodes and weights that we use in this paper.

```
import numpy as np
from numba import jit
@jit(nopython=True, cache=True)
def gauss_zeroes_weights(x1, x2, n):
    const_yeps = 3.0e-14
    x = np.zeros(n)
    w = np.zeros(n)
    m = int((n+1)/2)
    yxm = 0.5*(x2 + x1)
    yxl = 0.5*(x2 - x1)
    for i in range(m):
        yz = np.cos(np.pi*(i + 0.75)/(n + 0.5))
        while True:
            yp1 = 1.0
            yp2 = 0.0
            for j in range(n):
                yp3 = yp2
                yp2 = yp1
                yp1 = ((2.0*j + 1.0)*yz*yp2 - j*yp3)/(j+1)
            ypp = n*(yz*yp1 - yp2)/(yz*yz - 1.0)
            yz1 = yz
            yz = yz1 - yp1/ypp
            if (np.abs(yz - yz1) < const_yeps):
                break # exit while loop
        x[i] = yxm - yz*yxl
        x[n-1-i] = yxm + yxl*yz
        w[i] = 2.0*yxl/((1.0 - yz*yz)*ypp*ypp)
        w[n-1-i] = w[i]
    return x, w
```

Fig.6: Computation of `n` Gauss nodes within `x1:x2` interval.

Note a few peculiarities of Eq.(7). First, in Eq.(7), 2*N* is the order of the Gaussian quadrature, where *N* is the number of nodes (often called "streams" in RT) per hemisphere. Since nodes are located symmetrically with respect to μ=0 (the horizon), it is common in RT to define *N* (half space), not 2*N* (full space) as the input parameter.

Second, there are two types of Gaussian quadrature: single and double [82]. Suppose, μ_1 and w_1 are *N* single-Gauss nodes, i.e. nodes computed in the [-1 : +1] (full-) interval, and μ_2 and w_2 are double Gauss nodes computed within [0 : +1] (half-) interval. The two are related as follows

$$\mu_2^+ = \frac{\mu_1 + 1}{2} > 0, \quad w_2^+ = \frac{w_1}{2}. \tag{8}$$

Fig.7 shows a full set of 2*N* double-Gauss nodes within [-1 : 1], defined using $\mu_2^- = -\mu_2^+ < 0, \quad w_2^- = w_2^+$, and a corresponding set of 2*N* single-Gauss nodes for 2*N*=20. Note that at the horizontal plane (μ = 0), weights of the double-Gauss nodes are smaller, so is the influence of discontinuity of intensity at the boundaries (e.g., atmosphere-surface boundary, or two adjacent atmospheric layers with different optical properties like cloud and clear sky). In contrast, single-Gauss quadrature seems preferable for computations of the phase function expansion moments due to higher density of nodes at exact forward (μ=+1) and backward (μ=-1) scattering directions.

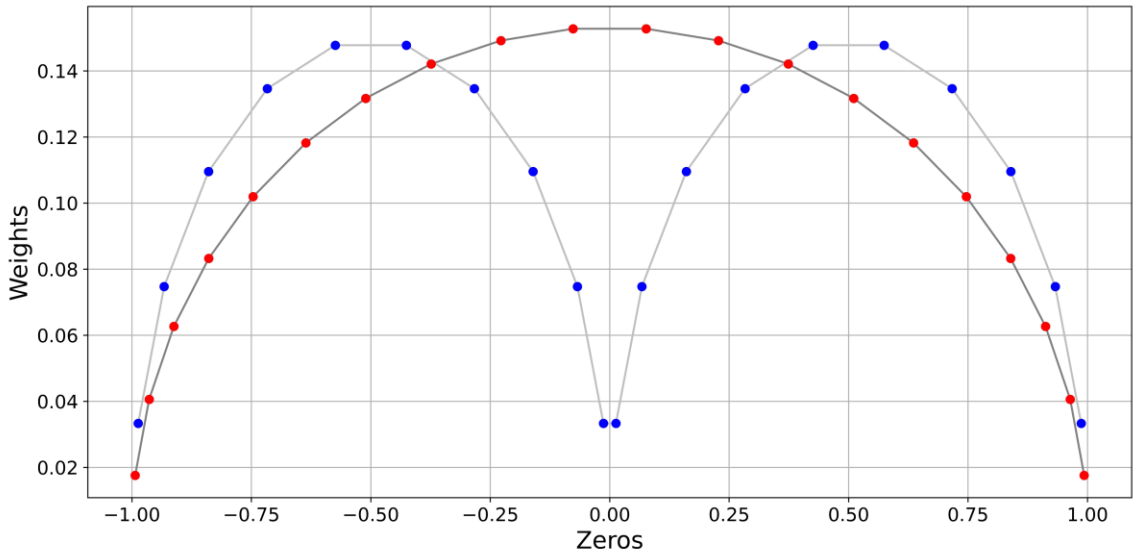


Fig.7: Single- (red) and double- (blue) Gaussian quadrature for 2*N*=20 nodes in the full interval

3.2. Basis functions in RT

Legendre polynomials are orthogonal polynomials and considered as the basis functions in RT numerical simulations. Depending on the problem, either ordinary, or associated, or generalized (for polarization) polynomials are used. Because they are orthogonal they can be added, replacing an integral with a simpler summation, where the number of terms can be adjusted as required to balance numerical accuracy with computational burden. In this section, we discuss computation of ordinary and associated Legendre polynomials separately.

3.2.1. Ordinary Legendre polynomials

Ordinary Legendre polynomials $P_k(x)$, where x=[-1 : 1] corresponds to cosine of the zenith angle, possess only one degree of freedom – the polynomial order k = 0, 1, 2, … K. Hence, they are used to compute solutions that depend on one angle only – zenith or scattering. Examples are computation of fluxes and azimuthally independent intensity (when either solar or view directions, or both, coincide with local normal), and expansion of the phase function. $P_k(x)$ are orthogonal

$$\int_{-1}^{1} P_k(x)P_l(x)dx = \frac{2}{2k+1}\delta_{kl}\,, \tag{9}$$

where δ_{kl} is the Kronecker delta. For k = 0 and 1, the Legendre polynomials are known: $P_0(x)=1$, $P_1(x)=x$. For arbitrary k, one uses the recurrence relation

$$(k+1)P_{k+1}(x) = (2k+1)\,x\,P_k(x) - k\,P_{k-1}(x)\,. \tag{10}$$

In RT codes, it is convenient to redefine indices, $k \rightarrow k$ -1, in Eq.(10), scale both sides by 1/k, and write

$$P_k(x) = (2-1/k)xP_{k-1}(x) - (1-1/k)P_{k-2}(x)\,. \tag{11}$$

Despite the simplicity of Eq.(11), it is a good idea to check it against explicit formula for low orders, available e.g. in Wolfram[15] (see Eqs.(4-8)) or Wikipedia[16] or literature [83] or built-in functions in Python (`scipy.special.legendre`), Matlab (`legendreP`) etc. **Fig.8** shows our implementation in Python.

```
1.  import numpy as np
2.  from numba import jit
3.  @jit(nopython=True, cache=True)
4.  def legendre_polynomial(x, kmax):
5.      nk = kmax+1
6.      pk = np.zeros(nk)
7.      if kmax == 0:
8.          pk[0] = 1.0
9.      elif kmax == 1:
10.         pk[0] = 1.0
11.         pk[1] = x
12.     else:
13.         pk[0] = 1.0
14.         pk[1] = x
15.         for ik in range(2, nk):
16.             pk[ik] = (2.0 - 1.0/ik)*x*pk[ik-1] - \
17.                      (1.0 - 1.0/ik)*pk[ik-2]
18.     return pk
```

Fig.8: Python function to compute ordinary Legendre polynomials for a given scalar value `x` and all orders `k = 0, 1, 2 … kmax`.

In our example `kmax` is defined with zero-offset for k = 0, 1, 2 … which gives the total of `kmax+1` values of k. The zero offset is convenient in e.g. C, Python, IDL, or other zero-based languages because "theoretical" value of k coincides with index of an element in array **pk**. However, in Fortran one should modify the above code as appropriate if a default unit offset for indices is used. The oscillatory nature of the polynomials may hide the error at this stage, if not tested properly, and will manifest itself in highly oscillating intensity or the phase function when computed as a function of angle.

Both the number of abscissae x and orders k may reach thousands for atmospheres with clouds, but a few tens to hundreds are more common for atmospheric aerosol studies. Although computation of the polynomials is never a bottleneck, the developer should decide on what parameter, k or x, will be in the lead (fast) dimension if a 2d-array is used to precompute and store $P_k(x)$. As we will see, summation over k is used to simulate multiple scattering. Therefore, it is beneficial to store all k for a given x consecutively in memory. The second option, sequence of x for a given k, is preferable for expansion of the phase function in Legendre series using integration over $x = \mu$ for a given k.

[15] http://mathworld.wolfram.com/LegendrePolynomial.html
[16] https://en.wikipedia.org/wiki/Legendre_polynomials

3.2.2. Associated Legendre polynomials

The associated Legendre polynomials[17,18] $P_k^m(x)$ possess 2 degrees of freedom, k and m, for zenith and azimuth expansions, respectively; m =0 reduces the associate polynomials to the ordinary ones. The numerical strategy is similar: one uses recurrence relation (e.g., see Eq.(6.8.7) in [80,81])

$$(k-m)P_k^m(x) = (2k-1)xP_{k-1}^m(x) - (k+m-1)P_{k-2}^m(x) \tag{12}$$

provided expressions for lower degree k and given order m are known. For all $m < k$, $P_k^m(x) = 0$ by definition due to m-th derivative of the k-order ordinary Legendre polynomial $P_k(x)$. The analytical expression is known for $k = m$ (e.g., see Eq.(6.8.8) in [80,81])

$$P_m^m(x) = (-1)^m(2m-1)!!(1-x^2)^{m/2}. \tag{13}$$

Eq.(12) and explicit formulae for low-order polynomials (e.g., Eqs. (12)-(27) in Wolfram[19] or in [83]) allow one to start iterations and check the result. Source code for $P_k^m(x)$ is published in C [81] and Fortran [80]. Built-in functions `scipy.special.lpmn`[20] and `legendre`[21] are available in Python and Matlab, respectively.

In this paper, however, we use a slightly different form of the associated polynomials (Schmidt semi-normalized polynomials[22], see also [84])

$$Q_k^m(x) = (-1)^m\sqrt{\frac{(k-m)!}{(k+m)!}}P_k^m(x) \tag{14}$$

for two reasons. First, the (-1)^m factor removes the same factor from $P_k^m(x)$ (e.g. see Eq.(13)). Second, the use of the square root factor simplifies the orthogonal property

$$\int_{-1}^{1} Q_k^m(x)Q_l^m(x) = \frac{2}{2k+1}\delta_{kl}, \tag{15}$$

where δ_{kl} is the Kronecker delta (compare e.g., with Eq.(5) in Wolfram[16]). Therefore, one must be careful when comparing associated polynomials from different sources. Note that the Schmidt and associate polynomials are direct substitution of each other in terms of the final result, but their orthogonal properties differ by a scaling factor. In **Table 2** we provide analytical expressions for a few low-order associated polynomials as defined in Eq.(14); **Fig.9** shows source code.

[17] https://mathworld.wolfram.com/AssociatedLegendrePolynomial.html
[18] https://en.wikipedia.org/wiki/Associated_Legendre_polynomials
[19] https://mathworld.wolfram.com/AssociatedLegendrePolynomial.html
[20] https://docs.scipy.org/doc/scipy/reference/generated/scipy.special.lpmn.html
[21] https://www.mathworks.com/help/matlab/ref/legendre.html
[22] https://en.wikipedia.org/wiki/Spherical_harmonics

Table 2: Explicit expressions for a few low-order associated polynomials $Q_k^m(x)$, Eq.(14)

k	m = 1	m = 2	m = 3
0	0	0	0
1	$\sqrt{(1-x^2)/2}$	0	0
2	$3x\sqrt{(1-x^2)/6}$	$3(1-x^2)/\sqrt{24}$	0
3	$3(5x^2-1)\sqrt{(1-x^2)/3}/4$	$15x(1-x^2)/\sqrt{120}$	$15(1-x^2)^{3/2}/\sqrt{720}$
4	$-\sqrt{5}x(3-7x^2)\sqrt{(1-x^2)}/4$	$15(7x^2-1)(1-x^2)/\sqrt{1440}$	$105x(1-x^2)^{3/2}/\sqrt{5040}$

```
import numpy as np
from numba import jit
@jit(nopython=True, cache=True)
def schmidt_polynomial (m, x, kmax):
    nk = kmax+1
    qk = np.zeros(nk)
#   k=m: Qmm(x)=c0*[sqrt(1-x2)]^m
    c0 = 1.0
    for ik in range(2, 2*m+1, 2):
        c0 = c0 - c0/ik
    qk[m] = np.sqrt(c0)*np.power(np.sqrt( 1.0 - x*x ), m)
#   Q{k-1}m(x), Q{k-2}m(x) -> Qkm(x)
    m1 = m*m - 1.0
    m4 = m*m - 4.0
    for ik in range(m+1, nk):
        c1 = 2.0*ik - 1.0
        c2 = np.sqrt((ik + 1.0)*(ik - 3.0) - m4)
        c3 = 1.0/np.sqrt((ik + 1.0)*(ik - 1.0) - m1)
        qk[ik] = (c1*x*qk[ik-1] - c2*qk[ik-2])*c3
    return qk
```

Fig.9: Python function to compute Schmidt semi-normalized associated Legendre polynomials, Eq.(14), for a given scalar value *x*, Fourier degree $m > 0$, and all orders $k = 0, 1, 2 \ldots$ kmax. Note for $m < k$, $Q_k^m(x) = 0$ is returned.

3.3. Solution in the single scattering approximation

The single scattering approximation is a particular but very important solution to the RTE. Recall it is derived from Eq.(1) omitting the (multiple) scattering integral $J(\tau,\mu,\varphi)$, Eq.(2), to give

$$\mu\frac{\partial I_1(\tau,\mu,\varphi)}{\partial\tau} = -I_1(\tau,\mu,\varphi) + \frac{\omega_0}{4\pi}p(\mu,\mu_0,\varphi,\varphi_0)I_0\exp\left(-\frac{\tau}{\mu_0}\right), \qquad (16)$$

where subscript "1" means single scattering (as opposed to superscript denoting Fourier index *m*). Note, Eq.(16) contains only path radiance because bouncing of (already scattered) light between atmosphere and surface requires at least two scattering events (reflection is equivalent to scattering in that regard).

The surface contributes to single scattering going upward only by reflecting the direct (not yet scattered) solar beam

$$I_1^{Atm+Srf} = I_1 + I_1^{Srf} = I_1 + I_0 \exp(\tau_0/\mu_-)\rho\exp(-\tau_0/\mu_0)\,, \tag{17}$$

Eq.(16) is an ordinary differential equation. The solution $I_1(\tau,\mu,\varphi)$ obeys two boundary conditions, at TOA (descending diffuse radiation is zero) and BOA (ascending diffuse radiation is zero even if the surface is not black). Because of that, the solution in single scattering looks different for ascending, $\mu < 0$, and descending, $\mu > 0$, radiation

$$I_1(\tau,\mu,\varphi) = I_1(\tau,\Theta) = \frac{\omega_0}{4\pi}\,p(\Theta)\begin{cases} \dfrac{\mu_0}{\mu_0-\mu}\left[\exp\left(-\dfrac{\tau}{\mu_0}\right)-\exp\left(\dfrac{\tau_0-\tau}{\mu}-\dfrac{\tau_0}{\mu_0}\right)\right], & \mu<0 \\ \dfrac{\mu_0}{\mu_0-\mu}\left[\exp\left(-\dfrac{\tau}{\mu_0}\right)-\exp\left(-\dfrac{\tau}{\mu}\right)\right], & \mu>0,\ \mu\neq\mu_0 \\ \dfrac{\tau}{\mu_0}\exp\left(-\dfrac{\tau}{\mu_0}\right), & \mu=\mu_0. \end{cases} \tag{18}$$

The last equation analytically eliminates the $1/(\mu_0-\mu)$ singular point, which also can be avoided numerically by slightly shifting μ.

Eq.(18) describes general features of the angular distribution of the scattered light, and can be used to quickly compute derivatives over the parameters (e.g., optical thickness). Due to its simple analytical representation, it is used to enhance the code's performance as a routine known as *single scattering correction*.

Eq.(18) is inconvenient if $p(\Theta) = p(\tau,\Theta)$. In this case one splits the atmosphere into homogeneous layers $\Delta\tau$, computes single scattering from each layer, assumed alone $I_{11}(\tau,\Theta)$ ($\tau_0=\Delta\tau$ and $\tau=0$ or $\Delta\tau$ for upward and downward solutions, respectively),

$$I_{11}(\tau,\Theta) = \frac{\omega_0}{4\pi}\,p(\Theta)\begin{cases} \dfrac{\mu_0}{\mu_0-\mu}\left[1-\exp\left(\dfrac{\Delta\tau}{\mu}-\dfrac{\Delta\tau}{\mu_0}\right)\right], & \mu<0 \\ \dfrac{\mu_0}{\mu_0-\mu}\left[\exp\left(-\dfrac{\Delta\tau}{\mu_0}\right)-\exp\left(-\dfrac{\Delta\tau}{\mu}\right)\right], & \mu>0,\ \mu\neq\mu_0 \\ \dfrac{\Delta\tau}{\mu_0}\exp\left(-\dfrac{\Delta\tau}{\mu_0}\right), & \mu=\mu_0 \end{cases} \tag{19}$$

and then computes the contribution from that layer to all other layers above and below the current one, accounting for the exponential attenuation along the view direction as appropriate. The solar beam must also be attenuated. **Fig.10** illustrates the idea. By looping over all $\Delta\tau$-layers, one accumulates the contribution from each and thus gets the single scattering solution at all levels in the atmosphere, which we will use further for multiple scattering computations.

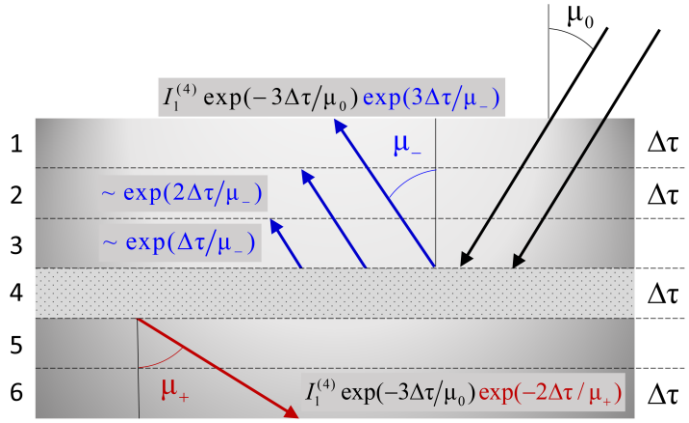


Fig.10 Single scattering from layer 4, $I_1^{(4)}$ see Eq.(19), irradiated by an attenuated direct solar beam contributes to each boundary above and below the layer. The contribution is also attenuated after scattering depending on the view cosine. Accumulated contribution from all layers 1 to 6 yields single scattering at each boundary of the layer.

For validation purposes, we recommend implementing two versions of the single scattering solution. The one given by Eq.(18) is easier to implement: it does not loop over element layers, and attenuation along the solar beam and the view direction are accounted for in the equation "automatically". Eq.(19) is practically convenient but requires more coding – hence the chance of making a mistake is higher. Benchmark results are published [85] for single scattering of polarized light, which in the case of solar light precisely agrees with the scalar approximation made here. **Fig.11(a-b)** shows our implementation of Eq.(18) separately for upward (`single_scattering_up`) and downward (`single_scattering_down`) radiation. Source code for Eq.(19) will be shown later.

```
import numpy as np
from numba import jit
from def_legendre_polynomial import legendre_polynomial
@jit(nopython=True, cache=True)
def single_scattering_up(mu, mu0, azr, tau0, xk):
    nk = len(xk)
    smu = np.sqrt(1.0 - mu*mu)
    smu0 = np.sqrt(1.0 - mu0*mu0)
    nu = mu*mu0 + smu*smu0*np.cos(azr)
    p = np.zeros_like(nu)
    for inu, nui in enumerate(nu):
        pk = legendre_polynomial(nui, nk-1)
        p[inu] = np.dot(xk, pk)
    mup = -mu
    I1up = p*mu0/(mu0 + mup)*(1.0 - np.exp(-tau0/mup -tau0/mu0))
    return I1up
```

Fig.11a: Python code for Eq.(18), ascending radiation: `mu` < 0; array of azimuths **azr** is in radians. Expansion moments **xk** are scaled by half of the single scattering albedo – see Section 3.5 "TOA flux and scaling of the phase function and moments".

```
1.  import numpy as np
2.  from numba import jit
3.  from def_legendre_polynomial import legendre_polynomial
4.  @jit(nopython=True, cache=True)
5.  def single_scattering_down(mu, mu0, azr, tau0, xk):
6.      nk = len(xk)
7.      tiny = 1.0e-8       # for 1/(mu-mu0) singular point
8.      smu = np.sqrt(1.0 - mu*mu)
9.      smu0 = np.sqrt(1.0 - mu0*mu0)
10.     nu = mu*mu0 + smu*smu0*np.cos(azr)
11.     p = np.zeros_like(nu)
12.     for inu, nui in enumerate(nu):
13.         pk = legendre_polynomial(nui, nk-1)
14.         p[inu] = np.dot(xk, pk)
15.     if np.abs(mu - mu0) < tiny:
16.         I1dn = p*tau0*np.exp(-tau0/mu0)/mu0
17.     else:
18.         I1dn = p*mu0/(mu0 - mu)* \
19.                (np.exp(-tau0/mu0) - np.exp(-tau0/mu))
20.     return I1dn
```

Fig.11b: Same as **Fig.11a** except for descending radiation: `mu` > 0. Expansion moments **xk** are scaled by half of the single scattering albedo – see Section 3.5 "TOA flux and scaling of the phase function and moments".

In both functions `single_scattering_up` and `single_scattering_down`, the phase function is represented using expansion moments, **xk** (scaled by half of single scattering albedo – see Section 3.5). Next section discusses this key step in RT numerical simulation.

3.4. Expansion of the phase function in Legendre series, and addition theorem

In both code samples for the upward and downward single scattering intensity, **Fig.11ab**, one finds expansion of the phase function in Legendre series using dot product

```
import numpy as np
…
pk = legendre_polynomial(nu, nk-1)
p = np.dot(xk, pk)
```

for which the analytical formula is

$$p(\nu) = \sum_{k=0}^{K} x_k P_k(\nu)\,. \tag{20}$$

This expansion can be omitted in the case of single scattering, but for multiple scattering it is a key step. The reason for that is the cosine of the scattering angle (Θ)

$$\nu = \cos\Theta \tag{21}$$

is not a direct variable of the RTE. However, it is indirectly related to the RTE variables through the following:

$$\nu = \mu\mu' + \sqrt{1-\mu^2}\sqrt{1-\mu'^2}\cos(\varphi - \varphi') \tag{22}$$

In Eq.(22), the prime notation is used for radiation before scattering, which direction is arbitrary and unknown, except for the incoming Solar beam: $\mu' = \mu_0$, $\varphi' = \varphi_0 = 0$.

The addition theorem for Legendre polynomials combines Eqs.(20) and (22) using ordinary and associated Legendre polynomials:

$$p(\nu) = p(\mu, \mu', \varphi, \varphi') = p^0 + 2\sum_{m=1}^{M} p^m \cos(m(\varphi - \varphi')) =$$

$$= \sum_{k=0}^{K} x_k P_k^0(\mu) P_k^0(\mu') + 2\sum_{k=0}^{K} \sum_{m=k}^{M<K} x_k P_k^m(\mu) P_k^m(\mu') \cos(m(\varphi - \varphi')) \tag{23}$$

Expansion Eq.(23) (in short form Eq.(6)) contains Fourier series, whose azimuthally averaged component p^0 we write as a separate term for convenience. Using the Fourier expansion of intensity, Eq.(5), and orthogonal properties of the cosine function under the scattering integral, one gets the azimuthally-independent (decoupled) equation for the *m*-th component of the scattered intensity (we use integral form of the RTE as an example)

$$\mu \frac{\partial I^m(\tau, \mu)}{\partial \tau} + I^m(\tau, \mu) = \frac{\omega_0}{2} \int_{-1}^{1} I^m(\tau, \mu') p^m(\mu, \mu') d\mu' + \frac{\omega_0}{4\pi} I_0 p^m(\mu, \mu_0) \exp\left(-\frac{\tau}{\mu_0}\right) \tag{24}$$

The expansion moments of the phase function are a key input parameter for multiple scattering simulations. At this step we assume them known and will briefly discuss their computation later in Section 4.3. Before we proceed to Section 3.6 that shows sample code that solves Eq.(24) using the method of GS iterations (the code is named `gsit` as a result), we need to define boundary conditions in Section 3.5.

3.5. TOA flux, scaling of the phase function and the bottom boundary condition

In Section 2.1 we mentioned that the scaling of the RTE solution is in general arbitrary. For our code we pick this value of the TOA flux:

$$I_0 = 2\pi \tag{25}$$

which converts Eq.(24) to (note the single scattering term – last in the right side - has changed)

$$\mu \frac{\partial I^m(\tau, \mu)}{\partial \tau} + I^m(\tau, \mu) = \frac{\omega_0}{2} \int_{-1}^{1} I^m(\tau, \mu') p^m(\mu, \mu') d\mu' + \frac{\omega_0}{2} p^m(\mu, \mu_0) \exp\left(-\frac{\tau}{\mu_0}\right). \tag{26}$$

Further, we note that the scattering event is always described by multiplication of the single scattering albedo ω_0 and the phase function (in single scattering) or moments (in the Fourier expanded single and multiple scattering). To reduce the number of arguments in corresponding functions, we include $\omega_0/2$ as a scaling factor in the phase function moments and deal with this equation (note $\omega_0/2$ is "dropped")

$$\mu \frac{\partial I^m(\tau, \mu)}{\partial \tau} + I^m(\tau, \mu) = \int_{-1}^{1} I^m(\tau, \mu') p^m(\mu, \mu') d\mu' + p^m(\mu, \mu_0) \exp\left(-\frac{\tau}{\mu_0}\right), \tag{27}$$

where the k = 0 expansion moment for the phase function becomes $x_0 = \omega_0/2$, and other moments are scaled by the same factor. It is for that reason the single scattering functions in **Fig.11ab** do not explicitly take ω_0 as an input parameter.

Definition of the TOA flux, Eq.(25), modifies the bottom boundary condition, Eq.(3), - the factor of 2 in reflection of the direct solar beam (first term in the right side) appears

$$I(\tau_0,\mu_-,\varphi) = 2\rho\mu_0 \exp\left(-\frac{\tau_0}{\mu_0}\right) + \frac{\rho}{\pi}\int_0^{2\pi}\int_0^1 \mu\, I(\tau_0,\mu,\varphi) d\mu\, d\varphi\,. \tag{28}$$

The multiple bouncing (last) term in Eq.(28) contains doubled reflected intensity averaged over azimuth (m=0 Fourier harmonic as the superscript in $I^0(\tau_0,\mu)$ indicates)

$$\frac{1}{\pi}\int_0^{2\pi} I(\tau_0,\mu,\varphi)\, d\varphi = \frac{2}{2\pi}\int_0^{2\pi} I(\tau_0,\mu,\varphi)\, d\varphi = 2I^0(\tau_0,\mu) \tag{29}$$

Thus, the bottom boundary condition

$$I(\tau_0,\mu_-,\varphi) = I^0(\tau_0,\mu_-) = 2\rho\mu_0 \exp\left(-\frac{\tau_0}{\mu_0}\right) + 2\rho\int_0^1 \mu\, I^0(\tau_0,\mu) d\mu \tag{30}$$

contains the factor of 2 at both terms, but for different reasons. In case of a Lambertian surface at BOA, only m=0 Fourier harmonics for diffuse intensity survives after reflection. Using the bottom boundary condition, Eq.(30), we now proceed to computation of multiple scattering.

3.6. Numerical simulation of multiple scattering using Gauss-Seidel iteration

The method of GS iterations is a simple yet powerful numerical technique to solve the RTE. Introduced in RT in the 1960s for application in optical [72] and radio [86] bands, the method is actively used for numerical simulation of scattered light and its derivatives in plane-parallel [36,62,87] and spherical [74,88–93] atmospheres. We refer an interested reader to the above papers and references therein for details of the method.

For our purposes, the GS method allows to illustrate the main steps of numerical implementation of the RTE solution with arguably simplest possible non-Monte Carlo code. Thus, for instance, polarization of light was included in the GS code from the very beginning; adding surface reflectance, including a rough wind-ruffled ocean [94], is also straightforward. The code does not require external libraries or understanding sophisticated eigenvalue decomposition formalism of the DO or SH methods. Noteworthy, the methods of GS iterations and SO are conceptually identical and understanding of one immediately leads to understanding of the other. Later in this section we will highlight a minor peculiarity that distinguishes the two techniques.

Despite the general similarity of the GS and SO techniques, the former seems to have numerical supremacy over the latter. Dave [95] showed that GS requires 40%-60% less iterations for computation of downward fluxes (Fourier moment m=0 only) with four significant figures (~0.1% error). Azimuthal dependence was not studied. In the same remarkable paper, Dave analyzed the influence of integration step $\Delta\tau$ = 0.005, 0.01, 0.02 on the final accuracy in Rayleigh and Mie-scattering scenarios. He concluded [95] that the method of GS possess "*more efficient iteration procedure*" and found $\Delta\tau \sim 0.01$ meets the

0.1% error requirement for Rayleigh and Mie scattering, wide range of optical depths and solar angles, while 0.02 "*give fairly reliable numerical result*" [95].

The idea of the method together with its strengths and weaknesses are narrated in [23] and [22]. One splits the atmosphere into layers of equal (for efficiency) optical thickness Δτ and solves the azimuthally decoupled RTE, Eq.(24). First, solution at Gauss nodes is obtained iteratively starting from the single scattering solution known at any point and direction in the atmosphere. Next the RTE is solved for arbitrary line of sight using either dummy node technique or integration of the source function technique (Section 3.6.2). The single scattering correction is applied to increase accuracy without significantly increasing runtime (or equivalently: decrease the runtime for the same given accuracy). In subsequent sections we provide details of the algorithm. For clarity, we split each function into pieces, comment each piece separately, and refer to **Fig.10** in support of our explanation.

3.6.1. Multiple scattering solution at Gauss nodes and a given Fourier order m

At this step, our goal is to compute the m-th Fourier moment of the diffuse light at Gauss nodes and at all levels τ_i in the atmosphere, including TOA and BOA. The function `gauss_seidel_iterations_m()` solves the problem using inputs m – Fourier moment: `m` = 0, 1, 2 … K_{max}-1 (the function does not check the value of `m`); `mu0` – cosine of the SZA; `srfa` – Lambertian surface albedo (reflection coefficient); `nit` - number of iterations; `ng1` – number of Gauss nodes per hemisphere; `nlr` – number of layer elements; `dtau` – optical thickness of each layer element (integration step over τ); as Section 3.5 explains, array **`xk`** contains the phase function expansion moments scaled by single scattering albedo $\omega_0/2$ and $(2k+1)$. The total optical thickness of the layer (which, for this vertically-homogeneous case, is the whole atmosphere) is `tau0=dtau*nlr`. For simplicity we define the total number of iterations `nit` manually (although this can be done automatically). Variables `m`, `nit`, `ng1`, `nlr` are integers, other variables are floats. Function `gauss_seidel_iterations_m()` uses the following dependences: `gauss_zeroes_weights()` to compute Gauss zeros and weights, `legendre_polynomial()` to compute ordinary Legendre polynomials, and `schmidt_polynomial()` to compute associated (Schmidt semi-normalized) Legendre polynomials (**Fig.12**).

```
1.  import numpy as np
2.  from numba import jit
3.  from def_gauss_zeroes_weights import gauss_zeroes_weights
4.  from def_legendre_polynomial import legendre_polynomial
5.  from def_schmidt_polynomial import schmidt_polynomial
6.  @jit(nopython=True, cache=True)
7.  def gauss_seidel_iterations_m(m, mu0, srfa, nit, ng1, nlr, dtau, xk)
```

Fig.12a: Header for function `gauss_seidel_iterations_m()` to solve the m-th component of the RTE at Gauss nodes and grid over τ.

We begin by defining some local parameters as indicated in **Fig.12b**.

```
def gauss_seidel_iterations_m(m, mu0, srfa, nit, ng1, nlr, dtau, xk):
…
8.   tiny = 1.0e-8                        # to compare floats
9.   nb = nlr+1                           # number of boundaries
10.  nk = len(xk)                         # number of expansion moments
11.  ng2 = ng1*2                          # number of Gauss nodes per sphere
12.  tau0 = nlr*dtau                      # total optical thickness
13.  tau = np.linspace(0.0, tau0, nb)     # embedding of boundaries (Fig.6)
```

Fig.12b: Local variable parameters in `gauss_seidel_iterations_m(); tau` is the equidistant grid of levels.

We compute Gauss nodes μ_g and weights first per hemisphere and then extend them symmetrically for the whole sphere. The first and second halves of each array correspond to ascending and descending radiation, respectively.

```
def gauss_seidel_iterations_m(m, mu0, srfa, nit, ng1, nlr, dtau, xk):
…
14. mup, w = gauss_zeroes_weights(0.0, 1.0, ng1) # mup > 0
15. mug = np.concatenate((-mup, mup))            # up: -mup,  down: +mup
16. wg = np.concatenate((w, w))
```

Fig.12c: Double-Gauss nodes **mug** and weights **wg**.

Ordinary (for $m = 0$) or associated (for $m > 0$ – azimuthal dependence) Legendre polynomials and Fourier moments of the phase function are then computed from expansion coefficients depending on the input Fourier moment m. The same Fourier moment of the phase function is computed for scattering from the solar direction to all Gauss quadrature directions to be used in single scattering.

```
def gauss_seidel_iterations_m(m, mu0, srfa, nit, ng1, nlr, dtau, xk):
…
17. pk = np.zeros((ng2, nk)) # Legendre polynomials
18. p = np.zeros(ng2)        # m-th moment of the phase function
19. if m == 0:
20.     pk0 = legendre_polynomial(mu0, nk-1)
21.     for ig in range(ng2):
22.         pk[ig, :] = legendre_polynomial(mug[ig], nk-1)
23.         p[ig] = np.dot(xk, pk[ig, :]*pk0)
24. else:
25.     pk0 = schmidt_polynomial(m, mu0, nk-1)
26.     for ig in range(ng2):
27.         pk[ig, :] = schmidt_polynomial(m, mug[ig], nk-1)
28.         p[ig] = np.dot(xk, pk[ig, :]*pk0)
```

Fig.12d: Computation of polynomials and the phase function for the given Fourier moment m.

One can now compute the m-th Fourier moment for single scattering at each boundary for subsequent integration over optical depth, and at all Gauss directions for subsequent integration over the zenith. The latter is the lead dimension, for which elements are in memory with unit stride, for efficient integration over angle at each boundary.

We start computation of single scattering for descending radiation, the 2nd and 3rd parts of Eq.(19), by first computing single scattering from an element layer $\Delta\tau$, accumulating contributions for all element layers but considering the attenuation of the solar beam as it reaches each element layer and

attenuation along the line of sight. Recall, downward single scattering obeys the zero top boundary condition, while the bottom boundary has no effect (because for downward light in single scattering, the BOA has not yet been encountered). A numerical singularity is avoided by using Taylor series expansion (last formula in Eq.(19)).

```
def gauss_seidel_iterations_m(m, mu0, srfa, nit, ng1, nlr, dtau, xk):
…
29. I11dn = np.zeros(ng1)  # single scattering from element layer
30. for ig in range(ng1):
31.     mu = mup[ig]
32.     if (np.abs(mu0 - mu) < tiny):  # mu - mu0 -> 0 singularity
33.         I11dn[ig] = p[ng1+ig]*dtau*np.exp(-dtau/mu0)/mu0
34.     else:
35.         I11dn[ig] = p[ng1+ig]*mu0/(mu0 - mu)* \
36.                           (np.exp(-dtau/mu0) - np.exp(-dtau/mu))
37. I1dn = np.zeros((nb, ng1))  # single scattering at all levels ib
38. I1dn[1, :] = I11dn
39. for ib in range(2, nb):
40.     I1dn[ib, :] = \
41.         I1dn[ib-1, :]*np.exp(-dtau/mup) + \ # from boundary above
42.             I11dn*np.exp(-tau[ib-1]/mu0)    # from the current layer
```

Fig.12e: Solution to RTE in the single scattering approximation at positive Gauss nodes (2nd – lead – dimension) and all layer boundaries (first dimension).

Ascending single scattering is computed in a similar way, except for starting from BOA and going up. The only difference is the contribution of surface reflection of the direct solar beam. We limited ourselves to Lambertian surface with reflection that is azimuthally even – hence only the *m*=0 Fourier component is present for that surface.

```
def gauss_seidel_iterations_m(m, mu0, srfa, nit, ng1, nlr, dtau, xk):
…
43. I11up = p[0:ng1]*mu0/(mu0 + mup)* # single scattering from one layer
44.               (1.0 - np.exp(-dtau/mup -dtau/mu0))
45. I1up = np.zeros_like(I1dn)     # single scattering at all boundaries
46. if m == 0 and srfa > tiny:
47.     I1up[nb-1, :] = 2.0*srfa*mu0*np.exp(-tau0/mu0)          # surface
48.     I1up[nb-2, :] = I1up[nb-1, :]*np.exp(-dtau/mup) + \
49.                         I11up*np.exp(-tau[nb-2]/mu0)
50. else:
51.     I1up[nb-2, :] = I11up*np.exp(-tau[nb-2]/mu0)
52. for ib in range(nb-3, -1, -1):
53.     I1up[ib, :] = I1up[ib+1, :]*np.exp(-dtau/mup) + \
54.                       I11up*np.exp(-tau[ib]/mu0)
```

Fig.12f: Same **Fig.12e** except for negative (upward) Gauss directions. See Eq.[30] for scaling factor 2.0 in line 47.

We recommend checking this single scattering versus the one shown in **Fig.11ab**, at least for the azimuthally independent scenario, because the single scattering in **Fig.12(e-f)** will now be used to start iterations for multiple scattering.

Numerical simulation of multiple scattering involves the *m*-th Fourier moment of the phase function, computed from expansion moments, that simulates scattering from all Gauss directions to all Gauss

directions, Eq.(24). Basically, one deals with a square matrix that multiples a vector of intensities coming from all the Gauss directions.

The RTE scattering integral is replaced with matrix vector multiplication

$$\vec{\mathbf{J}} = \mathbf{W}\vec{\mathbf{I}} \tag{31}$$

where **W** is an `ng2`-by-`ng2` matrix (**wpij** in **Fig.12g**), $\vec{\mathbf{I}}$ is a vector of `ng2` intensities coming from all Gauss directions to the scattering point, and $\vec{\mathbf{J}}$ is a vector of intensities scattered towards all Gauss directions (not yet integrated over τ – hence we use notation **J** instead of **I**). Recall, upward directions (`mu` < 0) come first, followed by downward directions. With that in mind, we rewrite Eq.(31) as follows

$$\begin{bmatrix} \vec{\mathbf{J}}_- \\ \vec{\mathbf{J}}_+ \end{bmatrix} = \begin{bmatrix} \mathbf{W}_-^- & \mathbf{W}_+^- \\ \mathbf{W}_-^+ & \mathbf{W}_+^+ \end{bmatrix} \begin{bmatrix} \vec{\mathbf{I}}_- \\ \vec{\mathbf{I}}_+ \end{bmatrix} = \begin{bmatrix} \mathbf{W}_-^-\vec{\mathbf{I}}_- + \mathbf{W}_+^-\vec{\mathbf{I}}_+ \\ \mathbf{W}_-^+\vec{\mathbf{I}}_- + \mathbf{W}_+^+\vec{\mathbf{I}}_+ \end{bmatrix}. \tag{32}$$

In Eq.(32), all vectors and matrices are half the size, `ng1=ng2/2`, of those in Eq.(31), top and bottom indices at **W** correspond to sign of μ along row and column, respectively.

Several important notes must be made about the square matrix **`wpij`**. First, Eq.(32) reveals the meaning of each quarter of the scattering matrix **W**. Namely, $\mathbf{W}_-^-$ describes how light that was traveling upward (subscript "-"), $\vec{\mathbf{I}}_-$, continues its path upward (superscript "-") after scattering. Hence, $\mathbf{W}_-^-$ is the diffuse upward transmittance, $\mathbf{T}_-$. $\mathbf{W}_+^-$ describes how light that was traveling down, $\vec{\mathbf{I}}_+$, gets redirected up by scattering. Hence, $\mathbf{W}_+^-$ is the diffuse upward reflectance, $\mathbf{R}_-$. Applying the same idea to the two terms of $\vec{\mathbf{J}}_+$ leads one to rewrite Eq.(32) as

$$\begin{bmatrix} \vec{\mathbf{J}}_- \\ \vec{\mathbf{J}}_+ \end{bmatrix} = \begin{bmatrix} \mathbf{T}_- & \mathbf{R}_- \\ \mathbf{R}_+ & \mathbf{T}_+ \end{bmatrix} \begin{bmatrix} \vec{\mathbf{I}}_- \\ \vec{\mathbf{I}}_+ \end{bmatrix}. \tag{33}$$

Second, for a homogeneous layer there is no "up" and "down" – its optical properties are the same: $\mathbf{T}_- = \mathbf{T}_+ = \mathbf{T}$ and $\mathbf{R}_- = \mathbf{R}_+ = \mathbf{R}$, so

$$\begin{bmatrix} \vec{\mathbf{J}}_- \\ \vec{\mathbf{J}}_+ \end{bmatrix} = \begin{bmatrix} \mathbf{T} & \mathbf{R} \\ \mathbf{R} & \mathbf{T} \end{bmatrix} \begin{bmatrix} \vec{\mathbf{I}}_- \\ \vec{\mathbf{I}}_+ \end{bmatrix}. \tag{34}$$

In terms of the piece of code in **Fig.12g**, one can say that scattering from Gauss direction `i` to `j` and vice versa are equal and depend only on mutual "location" of the two ordinates. This is known as *the principal of reciprocity* [27]. Hence, **wpij** must be symmetric.

Third, as Table 2 indicates, non-zero elements in summation over *k* start with *m* – not with 0 (line 58 in **Fig.12g**). However, the gain of time is minor.

```
def gauss_seidel_iterations_m(m, mu0, srfa, nit, ng1, nlr, dtau, xk):
…
55. wpij = np.zeros((ng2, ng2))
56. for ig in range(ng2):
57.     for jg in range(ng2):
58.         wpij[ig, jg] = wg[jg]*np.dot(xk, pk[ig, :]*pk[jg, :])
```

Fig.12g: Fourier moment *m* for the phase matrix **p** for scattering from Gauss node `i` to Gauss node `j` weighted with **wg** for integration over all Gauss directions in the scattering integral - hence the name **wpij**.

Fourth and final, it can be shown [75] that upward and downward elements of matrix **W = wpij** involve only odd and even orders *k*, respectively. Accounting for that would involve unnecessary complication of the code and discussion of equations, without any significant advantage in performance. We skip this step for now, but use symmetry of the matrix **W**, Eqs.(33)-(34), which are basic equation for adding, AD, IE, and MO methods.

```
def gauss_seidel_iterations_m(m, mu0, srfa, nit, ng1, nlr, dtau, xk):
…
59. T = wpij[0:ng1, 0:ng1].copy()    # T.flags.c_contiguous = True
60. R = wpij[0:ng1, ng1:ng2].copy() # R.flags.c_contiguous = True
```

Fig.12h: Diffuse transmittance, **T**, and reflectance, **R**, matrices which elements are stored in row-major (C-style).

Since **R** and **T** will be used for numerical integration (summation) iteratively, it is crucial for efficiency to make sure their elements are contiguous in memory. The diffuse reflectance and transmittance matrices and single scattering solution allow us to iteratively compute multiple scattering at all boundaries and Gauss directions. Recall, for simplicity, we define the number of iterations `nit` manually as an input parameter.

Each scattering act is simulated using Eq.(24)

$$J^m(\tau,\mu_\pm)=\frac{\omega_0}{2}\int_{-1}^{+1}p^m(\mu_\pm,\mu')I^m(\tau,\mu')d\mu'. \tag{35}$$

In Eq.(35), we don't know intensity at arbitrary optical thickness τ, however we do know it at the nodes of the τ-grid. Given that Δτ is sufficiently small, it is fair to say the intensity does not change much from one boundary to other and can be replaced with an average value

$$J^m(\tau,\mu_\pm)\approx\bar{J}^m(\mu_\pm)=\frac{\omega_0}{2}\int_{-1}^{+1}p^m(\mu_\pm,\mu')\frac{I^m(\tau_i,\mu')+I^m(\tau_{i\mp1},\mu')}{2}d\mu'. \tag{36}$$

This approximation has been widely used in GS and SO codes for `dtau`=0.01-0.02 [22,40,72]. Eq.(36) yields in the integral over τ (see Eq.(4), last term in the right-hand side)

$$\frac{1}{\mu_\pm}\int_{\Delta\tau}J^m(\tau,\mu_\pm)\exp\left(-\frac{\tau}{\mu_+}\right)dt\approx\bar{J}^m(\mu_\pm)\left[1-\exp\left(-\frac{\Delta\tau}{\mu_+}\right)\right]. \tag{37}$$

Eq.(37) simulates "re-scattering" of the scattered radiation. To compute intensity at the given level, one must add single scattering within Δτ and diffuse radiation from the previous level that reaches the current level without scattering. **Fig.12i** shows the corresponding piece of code for downward radiation.

```
def gauss_seidel_iterations_m(m, mu0, srfa, nit, ng1, nlr, dtau, xk):
…
61.  Iup = np.copy(I1up) # initialize iterations …
62.  Idn = np.copy(I1dn) # … with single scattering
63.  for itr in range(nit):
64.      Iup05 = 0.5*(Iup[0, :] + Iup[1, :])
65.      Idn05 = 0.5*(Idn[0, :] + Idn[1, :]) # TOA boundary: Idn[0, :]=0
66.      J = np.dot(R, Iup05) + np.dot(T, Idn05)
67.      Idn[1, :] = I11dn + (1.0 - np.exp(-dtau/mup))*J
68.      for ib in range(2, nb):
69.          Iup05 = 0.5*(Iup[ib-1, :] + Iup[ib, :])
70.          Idn05 = 0.5*(Idn[ib-1, :] + Idn[ib, :])
71.          J = np.dot(R, Iup05) + np.dot(T, Idn05)
72.          Idn[ib, :] = Idn[ib-1, :]*np.exp(-dtau/mup) + \
73.                                  I11dn*np.exp(-tau[ib-1]/mu0) + \
74.                                      (1.0 - np.exp(-dtau/mup))*J
```

Fig. 12i: Implementation of Eqs.(35) and (37) for iterative computation of multiple scattering for descending radiation.

The array that describes downward radiation, **Idn**, is redefined in the `ib`−loop (**Fig.12i**: line 72). Once the loop is complete for `itr=0` (first iteration), **Idn** will contain single scattering (initializes iterations), re-scattered again (**Fig.12i**, line 66) and integrated over tau (**Fig.12i**, loop in line 68) – this is the two orders of scattering approximation.

To validate this intermediate result for two orders of scattering and catch possible errors at the first iteration, one can use a reliable benchmark [85] with input defined in that paper. In the aforementioned benchmark, the polarization is accounted for, and its influence on the total intensity is the strongest in the second scattering. However, 1) the vector benchmark is defined for aerosol layer of total optical thickness τ_0=0.2, not Rayleigh scattering, where polarization gives ~10% difference at τ_0~1.0 [96], and 2) the vector benchmark is still useful for catching errors in array indexing, and typos like using `mu` instead of `mu0` etc., which can often lead to numerical errors exceeding 10%. In addition to that, using vector successive orders RT code SORD [40] we have reproduced this benchmark with (vector mode) and without polarization (scalar mode) and found that influence of polarization was at the level of numerical error of SORD, ~0.1%.

Note **Idn** contains both scattering orders, not the second order only. In order to write a code based on the SO method, one has to store the result of line 72 in an array separate from the one that accumulates orders, instead of redefining **Idn**, and use only the second order (not the sum of the two) to compute the third one. This is basically the only feature that distinguishes computational procedures of the two algorithms, GS and SO.

The effect of array redefinition, as opposed to splitting by orders, can be seen in first iteration for computation of the upward radiation – **Fig.12j**.

```
def gauss_seidel_iterations_m(m, mu0, srfa, nit, ng1, nlr, dtau, xk):
…
63. for itr in range(nit):
    …
75.     if m == 0 and srfa > tiny:
76.         Iup[nb-1, :] = 2.0*srfa*np.dot(Idn[nb-1, :], mup*w) + \
77.                                2.0*srfa*mu0*np.exp(-tau0/mu0)
78.     Iup05 = 0.5*(Iup[nb-2, :] + Iup[nb-1, :]) # BOA: Iup[nb-1, :]=0
79.     Idn05 = 0.5*(Idn[nb-2, :] + Idn[nb-1, :])
80.     J = np.dot(T, Iup05) + np.dot(R, Idn05)
81.     Iup[nb-2, :] = Iup[nb-1, :]*np.exp(-dtau/mup) + \
82.                           I11up*np.exp(-tau[nb-2]/mu0) + \
83.                               (1.0 - np.exp(-dtau/mup))*J
84.     for ib in range(nb-3, -1, -1):            # -1 to include TOA
85.         Iup05 = 0.5*(Iup[ib, :] + Iup[ib+1, :])
86.         Idn05 = 0.5*(Idn[ib, :] + Idn[ib+1, :])
87.         J = np.dot(T, Iup05) + np.dot(R, Idn05)
88.         Iup[ib, :] = Iup[ib+1, :]*np.exp(-dtau/mup) + \
89.                             I11up*np.exp(-tau[ib]/mu0) + \
90.                                 (1.0 - np.exp(-dtau/mup))*J
91. return mug, wg, Iup[:, :], Idn[:, :]
```

Fig.12j: Same as **Fig.12i**, except for upward integration over τ; see Eq.(30) for the factor of 2.0 in lines 76 and 77.

At `itr=0`, in line 78, the upward intensity **`Iup`** is equal to that of single scattering **`I1up`** because it has not yet been redefined. However, in the next line 79, double scattering is used as computed by code in **Fig.12i**. Thus, in line 80, the code uses a "hybrid" light field: two scattering orders down, but only one up. This prevents one from using the mentioned benchmark [85] to check the upward direction unless the developer implements the method of two orders of scattering before proceeding with GS iteration. This strategy is highly recommended because it provides an opportunity to check the results against a reliable benchmark, and at the same time learn the SO method, which does not deviate too much from the presented GS iteration code framework.

Once the loop over all iterations is complete, one gets the solution at Gauss nodes and all levels in the atmosphere for upward **`Iup`** and downward **`Idn`** multiple scattering. These arrays, along with Gauss weights **`wg`** and nodes **`mug`** (to avoid recomputing them again) are returned from `gauss_seidel_iterations_m()`. In the next section we will show how to compute *m*-th Fourier component of intensity at arbitrary (user defined) direction using RTE solution at Gauss nodes and τ-grid.

3.6.2. Multiple scattering solution at arbitrary direction

3.6.2.1. General comments

There are several ways to compute intensity at user-defined nodes once the solution at Gauss nodes has been determined. Nearest neighbor or other interpolation may be sufficient if a high-order quadrature is used. Inserting a dummy node [97], associated with zero weight, is the next easiest way to solve the RTE at arbitrary view direction. It is convenient to "attach" the dummy nodes to positive or negative Gauss nodes depending on the sign of the dummy node. This approach works sufficiently well for remote sensing that uses one view zenith angle (e.g., AERONET). When one deals with only one view direction and one or few dozens of ordinates, performance (which is linked to the number of nodes and

ordinates) does not suffer. A symmetric dummy node is usually added into Eqs.(31)-(32), even if not used, to keep the size of "upward" and "downward" arrays even.

However, when one needs to generate a LUT, the number of dummy nodes becomes comparable or even exceeds the number of Gauss nodes. One can increase performance of integration over μ, simply by skipping dummy nodes in the summation. In this case, corresponding matrices would be rectangular to simulate scattering from all Gauss nodes (lead dimension e.g., row-major in C) to all Gauss nodes and user-defined nodes (column-wise). This strategy does not help much: iterations will spend time to solve RTE at dummy nodes which, because of their zero weight, contribute nothing back to the Gauss nodes. Nevertheless, the development of a dummy node version of an RT code could be useful for any selected numerical method of the RTE solution to check integration of the source function technique, described next.

The integration of the source function is identical to solution of the RTE at Gauss nodes, except no iterations are used. We know the RTE solution at Gauss nodes and all boundaries – hence we can evaluate the scattering integral from all Gauss directions to the user-defined one at any boundary and integrate it numerically, like we did for the Gauss nodes. The similarity between numerical integration over τ at Gauss nodes (iteratively) and at user-defined node (no iterations) makes this approach very convenient and easy to implement [98]. Integration of the source function is called in a loop over user-defined nodes independently, with each user-defined μ as an RTE parameter. It is wise to create two separate functions for upward and downward integration because they are a) slightly different due to boundary conditions and b) only one is required for most applications (except for rare synergetic retrievals of a scenes simultaneously observed from ground and space). The source function integration yields the solution at arbitrary level of the τ grid. But in most cases, solution at TOA (satellites) or BOA (ground-based sun photometers) is of interest. We now illustrate this technique for descending, `source_function_integrate_down()`, and ascending, `source_function_integrate_up()`, radiation.

3.6.2.2. Descending radiation: integration downward

Input for the downwards integration function contains Fourier moment *m*, the user's desired output downward direction `mu > 0`, solar angle `mu0`, number of layers `nlr`, step over tau `dtau`, the phase function expansion moments **xk**, Gauss nodes **mug** and weights **wg**, and intensity at all Gauss nodes in between each pair of boundaries, **Ig05** (see lines 64, 65 and 78, 79 in `gauss_seidel_iterations_m()`). Like downward single scattering, there is no surface parameter. The surface contributes via multiple scattering intensity at Gauss nodes. A few local parameters are defined like in `gauss_seidel_iterations_m()`, lines 8-13 in **Fig.13a**.

```
1. import numpy as np
2. from numba import jit
3. from def_legendre_polynomial import legendre_polynomial
4. from def_schmidt_polynomial import schmidt_polynomial
5. @jit(nopython=True, cache=True)
6. def source_function_integrate_down(m, mu, mu0, nlr, dtau, \
7.                                     xk, mug, wg, Ig05):
8.     tiny = 1.0e-8
9.     ng2 = len(wg)
10.    nk = len(xk)
11.    nb = nlr+1
12.    tau0 = nlr*dtau
13.    tau = np.linspace(0.0, tau0, nb)
```

Fig.13a: First few lines for integration of the source function downwards.

The function will compute single scattering from solar to user-defined direction (phase function **p**), and multiple scattering from Gauss directions to the user-defined one. Arrays with ordinary (m=0) or associated Legendre polynomials at solar (**pk0**), and Gauss (**pk**) directions and all necessary orders k are used for this purpose exactly like in `gauss_seidel_iterations_m()`, lines 17-18. A few new lines of code compute Legendre polynomials (**pku**) at the user-defined node – see **Fig.13b**.

```
def source_function_integrate_down(m, mu, mu0, nlr, dtau, \
                                   xk, mug, wg, Ig05):
    ...
14.    pk = np.zeros((ng2, nk))
15.    if m == 0:
16.        pk0 = legendre_polynomial(mu0, nk-1)
17.        pku = legendre_polynomial(mu, nk-1)
18.        for ig in range(ng2):
19.            pk[ig, :] = legendre_polynomial(mug[ig], nk-1)
20.    else:
21.        pk0 = schmidt_polynomial(m, mu0, nk-1)
22.        pku = schmidt_polynomial(m, mu, nk-1)
23.        for ig in range(ng2):
24.            pk[ig, :] = schmidt_polynomial(m, mug[ig], nk-1)
25.    p = np.dot(xk, pku*pk0)
```

Fig.13b: Computation of the polynomials at solar (`mu0`), user (`mu`), and Gauss (**mug**) nodes.

One then computes single scattering from an element layer `dtau` alone (`I1dn`), and a full single scattering solution (**I1dn**) at user-defined node only and nodes of the τ-grid – **Fig.13c**.

```
def source_function_integrate_down(m, mu, mu0, nlr, dtau, \
                                   xk, mug, wg, Ig05):
    ...
26.     if np.abs(mu - mu0) < tiny:
27.         I11dn = p*dtau*np.exp(-dtau/mu0)/mu0
28.     else:
29.         I11dn = p*mu0/(mu0 - mu)* \
30.                     (np.exp(-dtau/mu0) - np.exp(-dtau/mu))
31.     I1dn = np.zeros(nb)
32.     I1dn[1] = I11dn
33.     for ib in range(2, nb):
34.         I1dn[ib] = I1dn[ib-1]*np.exp(-dtau/mu) + \
35.                         I11dn*np.exp(-tau[ib-1]/mu0)
```

Fig.13c: Single scattering at the user node and all levels of τ.

Unlike in `gauss_seidel_iterations_m()` lines 55-58, the scattering matrix **wpij** is a vector (as `mu` is a single element rather than an array) that simulates scattering from all Gauss directions to the user-defined one – **Fig.13d**.

```
def source_function_integrate_down(m, mu, mu0, nlr, dtau, \
                                   xk, mug, wg, Ig05):
    ...
36. wpij = np.zeros(ng2)  # sum{xk*pk(mu)*pk(muj)*wj, k=0:nk}
37. for jg in range(ng2):
38.     wpij[jg] = wg[jg]*np.dot(xk, pku[:]*pk[jg, :])
```

Fig.13d: Scattering from all Gauss directions to a single user-defined direction.

As a final step, one accumulates the contribution from all element layers from TOA (zero boundary condition) to BOA – **Fig.13e**.

```
def source_function_integrate_down(m, mu, mu0, nlr, dtau, \
                                   xk, mug, wg, Ig05):
    ...
39. Idn = np.copy(I1dn)  # boundary condition: Idn[0, :] = 0.0
40. J = np.dot(wpij, Ig05[0, :])
41. Idn[1] = I11dn + (1.0 - np.exp(-dtau/mu))*J
42. for ib in range(2, nb):
43.     J = np.dot(wpij, Ig05[ib-1, :])
44.     Idn[ib] = Idn[ib-1]*np.exp(-dtau/mu) + \
45.                     I11dn*np.exp(-tau[ib-1]/mu0) + \
46.                         (1.0 - np.exp(-dtau/mu))*J
47. return Idn[nb-1] - I1dn[nb-1]
```

Fig.13e: Integration over τ for user-defined descending radiation. Multiple (second and higher) scattering solution is returned for the given Fourier component.

As a result, the array **Idn** contains all scattering orders' contribution to the solution to *m*-th Fourier component of the RTE at all boundaries and arbitrary downward direction. By subtracting the single scattering solution for a subsequent single scattering correction (Section 3.6.3 ahead), the function returns multiple (2+) scattering solution at BOA.

3.6.2.3. *Ascending radiation: integration upward*

The upward integration is slightly more complicated because of the possible surface contribution. In addition to those of `source_function_integrate_down()`, `source_function_integrate_up()` uses the following input parameters: `srfa` – Lambertian surface albedo, **`Igboa`** – intensity at Gauss nodes at BOA (output of `gauss_seidel_iterations_m()`), and one parameter changes the sign: `mu` < 0 for upward directions.

Local parameters, polynomials, the phase function, and multiple scattering vector of intensity at all boundaries are computed exactly like in `source_function_integrate_down()` (lines 7-24, 35-37), upward single scattering at user-defined node – as in `gauss_seidel_iterations_m()` (lines 43-53). So, we immediately proceed to integration of scattered radiation from BOA to TOA. This is shown in **Fig.14**, where line number 39 is inherited from corresponding line in `source_function_integrate_down()`, and positive `mup = -mu > 0` for convenience.

```
def source_function_integrate_up(m, mu, mu0, srfa, nlr, dtau,
                                 xk, mug, wg, Ig05, Igboa):
    ...
39.  Iup = np.copy(I1up)
40.  if m == 0 and srfa > 0.0:
41.      Iup[nb-1] = 2.0*srfa*np.dot(Igboa, mug[ng1:ng2]*wg[ng1:ng2]) +\
42.                  + 2.0*srfa*mu0*np.exp(-tau0/mu0)
43.  J = np.dot(wpij, Ig05[nb-2, :])
44.  Iup[nb-2] = Iup[nb-1]*np.exp(-dtau/mup) + \
45.                I11up*np.exp(-tau[nb-2]/mu0) + \
46.                    (1.0 - np.exp(-dtau/mup))*J
47.  for ib in range(nb-3, -1, -1):
48.      J = np.dot(wpij, Ig05[ib, :])
49.      Iup[ib] = Iup[ib+1]*np.exp(-dtau/mup) + \
50.                   I11up*np.exp(-tau[ib]/mu0) + \
51.                       (1.0 - np.exp(-dtau/mup))*J
52.  return Iup[nb-1] - I1up[nb-1]
```

Fig.14: Vertical integration at arbitrary line of sight. Note, mu < 0 is an input parameter, while mup = -mu > 0 is defined within `source_function_integrate_up()` and used for convenience; see Eqs.(30) for the scaling factor 2.0 in lines 41 and 42.

Like `source_function_integrate_down()`, `source_function_integrate_up()` returns *m*-th Fourier component for multiple (2+) scattering within the atmosphere, and multiple bouncing of light between the atmosphere and underlying reflecting surface, and only at TOA.

3.6.3. Main program: accumulation of the Fourier series

To combine and test these functions as a sample RT code, we consider two cases: Rayleigh scattering over a black (non-reflecting) surface, and fine aerosol (no Rayleigh) over a reflecting surface. For aerosol we used a volcanic Model 3 from [99]. Apart from the mentioned differences in the surface, the phase function and the number of Fourier moments, all other parameters are the same for simplicity. We validate the results against the DO RT code IPOL [41] in scalar (no polarization) mode and report runtime. First, we define solar-view geometry, single scattering albedo, and accuracy parameters as **Fig.15a** indicates.

```
Script: gsit.py
1.  import time
2.  import numpy as np
3.  from def_gauss_seidel_iterations_m import gauss_seidel_iterations_m
4.  from def_source_function_integrate_down import \
                                        source_function_integrate_down
5.  from source_function_integrate_up import \
                                        source_function_integrate_up
6.  from def_single_scattering_down import single_scattering_down
7.  from def_single_scattering_up import single_scattering_up
8.  #
9.  prnt_scrn = True  # print out intensities on the screen
10. fname_bmark = 'test_gsit_R&A_srf.txt' # file with benchmark data
11. phasefun = 'a'    # 'r': rayleigh; aerosol otherwise (case sensitive)
12. nit = 10          # number of Gauss-Seidel iterations
13. ng1 = 8           # number of Gauss nodes per hemisphere
14. nlr = 100         # number of layer elements
15. dtau = 0.01       # integration step over tau
16. ssa = 0.99999999 # single scattering albedo
17. sza = 45.0        # solar zenith angle, degrees
18. mup = np.linspace(-0.2, -0.9, num=8)  # cos(view zenith angle)
19. azd = np.array([0.0, 45.0, 90.0, 135.0, 180.0])  # relative azimuths
```

Fig.15a: RT code `gsit`: main program.

At this step, the view zeniths are defined via cosines and for TOA only. Below we define symmetric directions at BOA. Then comes the definition of the phase function expansion moments, desired number of the Fourier moments to be computed, surface reflectance, and location of column with benchmark results in the benchmark file. Note, the moments already contain the (2k+1) scaling factor. Hence, for the aerosol case [99], average scattering cosine is xk[1]/(2*1+1)= 2.084911/3 ≈ 0.695 (**Fig.15b**).

```
Script: gsit.py
    ...
20. if phasefun == 'r':
21.     print("Rayleigh:")
22.     nm = 3
23.     xk = np.array([1.0, 0.0, 0.5])
24.     srfa = 0.0  # surface albedo
25.     icol = 3
26. else:
27.     print("Aerosol:")
28.     nm = 10
29.     xk = np.array([1.000000, 2.084911, 2.459134, 2.234752, \
30.                    1.873098, 1.492956, 1.164725, 0.881976, \
31.                    0.689172, 0.506016, 0.402247, 0.289742, \
32.                    0.232980, 0.165427, 0.133290, 0.093127, \
33.                    0.074444, 0.050682, 0.039800, 0.025983, \
34.                    0.019929, 0.012274, 0.009257, 0.005312, \
35.                    0.004015, 0.002120, 0.001638, 0.000764, \
36.                    0.000602, 0.000211, 0.000178, 0.000033, \
37.                    0.000037, 0.000004, 0.000005, 0.000000])
38.     srfa = 0.3   # surface albedo
39.     icol = 4
```

Fig.15b: Input data for Rayleigh (see $\alpha_1^0, \alpha_1^1 = 0$, and α_1^2 [24]) and aerosol scattering (see column α_l^1 in Table 7 from [99]).

Computation of the cosine of the solar angle, downward user-defined nodes, azimuth in radians and lengths of arrays finalize the preparation step (**Fig.15c**)

```
Script: gsit.py
    ...
40. time_start = time.time()
41. mu0 = np.cos(np.radians(sza))    # cosine of solar zenith, mu0 > 0
42. mudn = -mup                      # cosines of downward view zeniths
43. nmu = len(muup)                  # number of upward view zeniths
44. azr = np.radians(azd)            # relative azimuths in radians
45. naz = len(azd)                   # number of relative azimuths
46. nrows = nmu*naz                  # number of rows in output array
```

Fig.15c: Auxiliary input parameters.

Now one computes exact single scattering (**Fig.15d**), including the TOA surface contribution. "Exact" relates to representation of the phase function: no Fourier series are involved and either all moments **xk** are used, or the phase function is interpolated over scattering angle to get values at the view direction. In the latter case, expansion is completely avoided in the single scattering correction.

```
Script: gsit.py
    ...
47. Itoa = np.zeros((nmu, naz))
48. if srfa > 0.0:
49.     for imu, mu in enumerate(muup):
50.         Itoa[imu, :] = single_scattering_up(mu, mu0, azr, \
51.                                             nlr*dtau, 0.5*ssa*xk) +\
52.                2.0*srfa*mu0*np.exp(-nlr*dtau/mu0)*np.exp(nlr*dtau/mu)
51. else:
52.     for imu, mu in enumerate(muup):
53.         Itoa[imu, :] = single_scattering_up(mu, mu0, azr, \
54.                                             nlr*dtau, 0.5*ssa*xk)
55. Iboa = np.zeros((nmu, naz))
56. for imu, mu in enumerate(mudn):
57.     Iboa[imu, :] = single_scattering_down(mu, mu0, azr, \
58.                                           nlr*dtau, 0.5*ssa*xk)
```

Fig.15d: This section computes exact single scattering path radiance at TOA and BOA. For TOA, contribution of the direct solar beam reflectance is accounted for. Refer to Eq.[30] for the factor of 2.0 in line 52 and to Sec.3.5 for the factor of `0.5*ssa` in lines 51, 54, and 58.

It is important to make sure that single scattering corresponds to the same incident TOA flux as the multiple scattering which is computed using Fourier series (loop over Fourier index *m*).

Now, for each azimuth order *m*, the RTE is solved at Gauss nodes and all boundaries for all scattering orders in **Fig.15e**.

```
Script: gsit.py
    ...
59. time_gsitm = 0.0
60. deltm0 = 1.0
61. for m in range(nm):
62.     t1 = time.time()
63.     mug, wg, Igup, Igdn = \
64.     gauss_seidel_iterations_m(m, mu0, srfa, nit, ng1, \
65.                               nlr, dtau, 0.5*ssa*xk)
66.     t2 = time.time()
67.     time_gsitm += t2 - t1
```

Fig.15e: Solve RTE for the given *m* at all Gauss nodes and optical depth grid. Refer to Sec.3.5 for the factor of 0.5*ssa in line 65.

Diffuse intensity for all Gauss directions, up and down, is averaged within each thin layer `dtau` (**Fig.15f**)

```
Script: gsit.py
    ...
61. for m in range(nm):
        ...
68.     Ig05 = np.zeros((nlr, 2*ng1))
69.     for ilr in range(nlr):
70.         Iup05 = 0.5*(Igup[ilr, :] + Igup[ilr+1, :])
71.         Idn05 = 0.5*(Igdn[ilr, :] + Igdn[ilr+1, :])
72.         Ig05[ilr, :] = np.concatenate((Iup05, Idn05))
```

Fig.15f: Diffuse intensity in between each pair of boundaries and at all Gauss directions.

These form input for functions that integrate along the line of sight (user-defined `mu`). Unlike `gauss_seidel_iterations_m()`, these functions return only multiple (second onward) scattering orders for the given Fourier moment *m*. This multiple scattering radiation is weighted with $\cos(m\varphi)$, Eq.(5) and added to the exact single scattering computed above (**Fig.15g**).

```
Script: gsit.py
    ...
61. for m in range(nm):
        ...
73. #   Accumulate Fourier series
74.     cma = deltm0*np.cos(m*azr)
75.     for imu, mu in enumerate(muup):
76.         Ims_toa = source_function_integrate_up(m, mu, mu0, srfa, \
77.                   nlr, dtau, 0.5*ssa*xk, mug, wg, Ig05, Igdn[nlr, :])
78.     Itoa[imu, :] += Ims_toa*cma
79.     for imu, mu in enumerate(mudn):
80.         Ims_boa = source_function_integrate_down(m, mu, mu0,
81.                   nlr, dtau, 0.5*ssa*xk, mug, wg, Ig05)
82.     Iboa[imu, :] += Ims_boa*cma
83.#
84.#     Kronecker delta = 2 for m > 0
85.     deltm0 = 2.0
86.     print('m =', m)
87.#end for m
```

Fig.15g: Integration along the line of sight and accumulation of the Fourier series finalize the RTE solution. Refer to Sec. 3.5 for the factor of `0.5*ssa` in lines 77 and 81.

Summation of the Fourier series completes the RTE solution. In the paper, we omit as trivial the part of the `gsit` script that reads the benchmark file and tests the code and print out the result. Instead, we refer the reader to the source code that comes with the paper. **Table 3** below indicates maximum and average errors vs benchmark for input parameters indicated above as well as the runtime for the whole script and separately for `gauss_seidel_iterations_m` (all Fourier orders combined). Appendix provides detailed output (`prnt_scrn = True` in line 9) and benchmark numbers (content of `test_gsit_R&A_srf.txt`, line 10).

Table 3: Accuracy (relative absolute error vs. benchmark IPOL) and runtime for the simulated cases[23].

Case	Max % error at TOA	Mean % error at TOA	Max % error at BOA	Mean % error at BOA	`gauss_seidel_iterations_m()` runtime, s.	`gsit` runtime, s.
Rayleigh	0.02				0.05	0.11
Aerosol	0.09	0.06	0.12	0.06	0.06	0.12

Not surprisingly, multiple scattering is the slowest part of the code, consuming 50% of the runtime (in fact more because the main script was not optimized with the high-performance Python compiler, Numba). In the following section we provide ideas to keep in mind while coding for high performance, further development, and support.

4. A Few Steps Forward

4.1. Mixing different types of scattering

In case of two or more scattering components, the RTE input is a linear mixture of scattering properties of each. Let us consider two scattering components, Rayleigh and absorbing aerosol, in presence of gas absorption. Assume Rayleigh layer is characterized by the total optical thickness, τ_R, phase function $p_R(\Theta)$, and expansion moments r_k, and aerosol is defined by the total aerosol thickness τ_A, phase function $p_A(\Theta)$, expansion moments α_k and single scattering albedo ω_A. The absorption optical thickness of the gas is τ_G. The phase functions and expansion moments are mixed linearly

$$p(\Theta) = w_R p_R(\Theta) + w_A p_A(\Theta); \quad x_k = \begin{cases} w_R r_k + w_A \alpha_k, & k = 0, 1, 2 \\ w_A \alpha_k, & k > 2 \end{cases} \tag{38}$$

where weights are defined by scattering optical thicknesses

$$w_R = \tau_R / (\tau_R + \omega_A \tau_A); \quad w_A = 1 - w_R \ . \tag{39}$$

The total optical thickness of the layer and single scattering albedo of the mixed components are

$$\tau = \tau_R + \tau_A + \tau_G; \ \ \omega_0 = (\tau_R + \omega_A \tau_A)/\tau \ . \tag{40}$$

[23] DELL Latitude E6520 laptop on Intel i7-2720QM CPU, 2.2 GHz, 8GB DD3 SDRAM (1333 MHz), Windows 10 64 bit; Python 3.7.6 integrated with Anaconda 3/Spyder 4 IDE.

Eqs.(38)-(40) are well known. However, regarding RT code development, three notes are worth mentioning: 1) to assure flexibility of the RT code, we recommend mixing components outside the RT solver; 2) Eq.(38) indicates that Rayleigh component affects only the first three moments, but higher moments from aerosol must be weighted as well; 3) adding Rayleigh smooths out the mixed phase function. Hence, numerically it might be better to mix the phase functions first and compute the expansion moments next, however the mixing ratio typically changes with height (known as the atmospheric profile). The next section discuss how RT code incorporate those.

4.2. Cases with variable vertical profiles

The presented version of `gsit` lacks one important feature: the dependence of optical properties on height (vertical profile). To include the vertical profiles would be a good exercise for an interested reader. This is not hard to do since GS and SO explicitly deal with numerical integration over τ, layer by layer. Hence, instead of layer-independent phase function and single scattering albedo, one deals with layer-dependent ones. In terms of expansion moments, scaled by single scattering albedo, one deals with **xk**`[0:nl, 0:nk]`, where Legendre order k is in the lead dimension for fast computation of the Fourier moment of the phase function. The number of optical layers, `nl`, is different from the number of steps `dtau`; the former may be a handful to a few dozen, while the latter may be dozens to hundreds. It is convenient to define a mask that relates indices of element layers `dtau` to a particular index of the optical layer. **Fig.16** shows an example for 2 optical layers and 9 element layers.

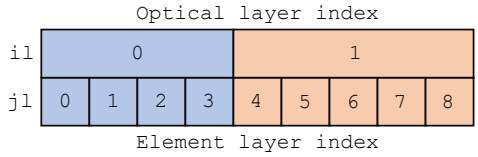


Fig.16a: Element layers 0-3 belong to the top optical layer, element layers 4-8 – to the bottom optical layer.

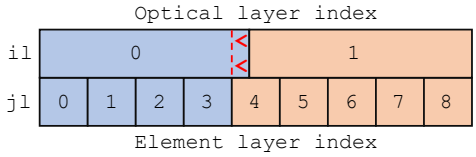


Fig.16b: Alternation of input vertical profile, `il=0:1`, to fit integer number of element layers `jl = 0:9` into each optical layer `il`.

The following simple code relates element and optical layers:

```
for jl in range(nlr):
    il = mask[jl]
    use xk[il, 0:nk]
```

It is of course unlikely that in application each optical layer will contain an integer number of element layers. In this case, as **Fig.16b** shows, one must slightly alter the input optical profile. One can find full Fortran texts of two subroutines that perform this task [38]. Validation of these subroutines [40] in case of realistic profiles [71] did not reveal problems caused by minor change of the input optical properties.

4.3. Computation and some properties of the expansion moments

Atmospheric optical properties often come from separate packages for light scattering by small particles. Many of those are publicly available and very carefully coded. In Mie codes [100–102] for light scattering by spherical particles, expansion moments **xk** are available together with the phase function. Usually, Mie codes are fast and can be built in the RT solver. T-matrix [103] and ray tracing codes are much slower. They are used for generating LUTs with some basic set of optical parameters (kernels). These LUTs require massive storage space: for example, kernel LUTs for spheroidal package [104] can be

as large as 5 GB. The LUTs are then interpolated (multidimensionally) to compute a specific user-defined case. Since this also takes time, it is recommended to precompute those "small" LUTs as well.

An important feature of non-Mie packages is that expansion moments are not readily available. One must compute those by numerically integrating the phase function (see Eq.(9)) using Gauss quadrature of sufficiently high order

$$x_k = \frac{2k+1}{2}\int_{-1}^{1} p(x)P_k(x)dx \approx \frac{2k+1}{2}\sum_{j=1}^{N} w_j\, p(x_j)P_k(x_j)\,. \tag{41}$$

A few remarks about Eq.(41): first, $x_0 = 1$, which never happens on practice due to numerical inaccuracy. This is avoided by scaling all elements by $1/x_0$. Second, unlike in `legpol()`, it is efficient to make the cosine of the scattering angle on the lead dimension. Third, as Eq.(41) indicates, the phase function must be computed at Gauss nodes, which is not always the standard output [104]. One must interpolate from the package grid to the Gauss nodes. This step requires some research: shall one interpolate in the angle space, or in the cosine space? Shall this be done on the log of the phase function (for smoothness)? What kind of interpolation should be used? Finally, one computes the expansion moments until either high-order moment become negligibly small, or when the phase function is represented at some scattering angle with sufficient accuracy. Given that every moment is computed with some error, and errors from all the moments are combined when one recalculates the phase function, the expansion must be approached with great care.

Unless it is necessary to combine the particle scattering code in RT solver, we recommend using it as a standalone tool and to precompute optical depths and phase functions in the form of LUTs. This not only saves time for RT simulation and makes it easier to debug potential problems, but also helps avoid hard-to-spot problems caused by using legacy and modern languages within the same packages (e.g. memory allocation).

4.4. BRDF

Accounting for azimuthally dependent BRDF is similar to what we have demonstrated above for the Lambertian case, except one has to expand BRDF in Fourier series over azimuth

$$\rho(\mu,\mu',\varphi) = \rho^0(\mu,\mu') + 2\sum_{m=1}^{M}\rho^m(\mu,\mu')\cos(m\varphi) \tag{42}$$

Symmetry of the BRDF with respect to principal plane is assumed (which is also a common assumption in Earth remote sensing) – hence only the cosine function is used in the series. Analytical integration over azimuth is replaced with numerical quadrature; Gaussian quadrature is usually used

$$\rho^0(\mu,\mu') = \frac{1}{2\pi}\int_0^{2\pi}\rho^0(\mu,\mu',\varphi)d\varphi = \int_0^1 \rho^0(\mu,\mu',x)dx \approx \sum_{j=1}^{N} w_j\rho^0(\mu,\mu',x_j) \tag{43}$$

$$\rho^m(\mu,\mu') = \frac{1}{2\pi}\int_0^{2\pi}\rho(\mu,\mu',\varphi)\cos(m\varphi)d\varphi \tag{44}$$

The order of the quadrature, N in Eq.(43), grows together with the specular component (e.g., glint) and in general estimated empirically. For ocean roughed by a 2 m/s wind, SORD uses 180 nodes for integration over azimuth [40].

There are two important things to mention. First, the BRDF reflects only those Fourier components that were generated in atmosphere. Thus, for instance, in a Rayleigh atmosphere, one needs only m = 0, 1, and 2. Higher Fourier components of the BRDF pass through atmosphere without scattering (and accounted on TOA by the single scattering correction). Second, one applies numerical Fourier expansion for each pair of incident-reflected zenith directions using the computationally expensive cosine function. There are two ways to accelerate the computations: 1) precompute and save the BRDF Fourier moments and 2) avoid multiple computations of cosines by expressing $\cos(m\,\varphi)$ as $\cos((m\text{-}1)\,\varphi + \varphi)$ and use trigonometric relations for the cosine of the sum[24]. This technique introduces dependence in RTE solution for different m, which is not an obstacle if one uses parallel computations above the solver level as we recommend in the next Section 4.5.

4.5. Suggestions for code efficiency and readability

RT codes (like other software) should be developed with efficiency and future support in mind. This includes many factors: on the one hand, it is reasonable to use the best algorithm for the given scenario, and a computationally efficient implementation of the algorithm. On the other hand, the time required to solve the problem is the sum of the code runtime and developing/debugging time. A simple but less runtime-efficient algorithm may turn out to be the best for the case. An example is the GS method for cloud remote sensing applications (i.e. high optical thickness). The DO and SH methods are more computationally efficient in this case (recall GS recalls division into layers of small optical thickness, while DO/SH runtime does not depend on optical thickness) but rely on a more complicated mathematical apparatus.

In this section we will discuss a few aspects related to the development of the `gsit` code presented here. There are many books and papers on efficient scientific software development, "Writing scientific software: a guide for good style" [105] and "The Software Optimization Cookbook" [106] are just two out of many examples. A brief introduction on "How to Write Fast Numerical Code" [107] is good to refresh one's memory on particular techniques for efficient coding. One can find short summaries on software engineering best practices in [108,109], [110]. In terms of coding philosophy, the online summary[25] of "Elements of Programming Style" [111] cannot be over recommended – especially "*make it right – then make it fast*".

Out of many useful recommendations in [112] and [113] we believe the most important for RT code development is optimization of memory reference. Elements of arrays should be located consecutively (unit-stride) to avoid "jumps". Thus, in arrays that contain data for numerical integration over angle and over optical thickness, the angle data is in the lead (fast) dimension as the angular integration happens at every boundary. C/C++ developers may decide to sacrifice clarity and go with one-dimensional arrays, instead of two-dimensional matrices, or use "fast matrix allocation method" [105]. Another advice is to avoid using computationally expensive functions, such as `np.exp(-dtau/mup)`, called multiple times in the innermost loop of `gauss_seidel_iterations_m()`. These quantities should

[24] https://ams.confex.com/ams/13CldPhy13AtRad/techprogram/paper_171283.htm
[25] https://en.wikipedia.org/wiki/The_Elements_of_Programming_Style

ideally be precomputed and called from memory. We have sacrificed that here for readability. Note that this option is available only for equal step Δτ=`dtau`.

Once the selected method of solution is coded sufficiently cleanly, further increase in performance is achieved by modifying the algorithm. As the previous section shows, `gauss_seidel_iterations_m()` is the slowest function that determines the runtime, t, of the whole code as:

$$t \sim M \cdot N_{\mathrm{IT}} \cdot N_{\mathrm{L}} \cdot N_{\mathrm{G}}^2 \ , \tag{45}$$

where M is the number of Fourier moments (equals the number of `gauss_seidel_iterations_m()` calls), N_{IT} is the number of iterations, N_{L} is the number of element layers Δτ, N_{G} is the number of gauss nodes. The latter is squared because one computes the contribution from all Gauss nodes to all Gauss nodes, basically, using matrices.

Obviously, faster performance can be achieved at the expense of accuracy by decreasing all parameters in Eq.(45). Alternatively, M can be reduced without loss of accuracy using a single scattering correction. In fact, it is now a standard option in all modern RT codes that simulate scattering of solar light (see "Introduction"). Since single scattering has an analytical representation, double scattering can be achieved by analytic integration of the single scattering over τ [85,114] – this will further reduce M. In the SO method, higher scattering orders require lower M due to smoothness of angular dependence of radiation scattered multiple times. We are not sure from our experience or aware of any publication showing that higher M can be computed using fewer iterations in the GS method. The "modified Fourier transform method" [95] utilize the idea that fewer M is needed if solar and/or view zenith is close to the local normal z (nadir or zenith). In the limit case of irradiation and/or observation along the z-axis (intensity is symmetric w.r.t. relative azimuth), m = 0 regardless the phase function.

The number of layers N_{L} can be decreased by replacing a simple technique, Eq.(37), with a more sophisticated rule that a) prevents precomputation of the exponential function and b) involves more costly functions such as square root. Indeed, the number of steps decrease, but the runtime may not [40]. Finally, the decrease of the number of Gauss ordinates is achieved by using the so-called "truncation" technique combined with single or double scattering correction [115].

Once everything possible has been squeezed from theoretical and numerical aspects, it is time for brute force: parallel computation using tools like MPI, OpenMP, for parallel and distributed processing, built-in Numba functions that translates Python to faster compiled machine code, and GPU processing. Independent solution of the RTE for a given Fourier order m looks like a great opportunity. In this paper, we solve the RTE for only one solar zenith angle (GS and SO methods), one wavelength, one set of atmospheric optical properties (optical thickness, phase function, single scattering albedo), and one surface. In applications, one runs the RTE solver for a large set of these. We therefore recommend running one RT solver on a single core and distribute scenarios, solar geometries, bands over many CPUs.

Readability is important. We recommend using code comments to describe all input/output parameters, including data type and size for arrays, in each function. If a global variable cannot be avoided for some reason, it must be described as well. Since RT codes are scientific software, developer may consider adding specific references (page and/or equation number) to papers and/or weblinks – as we did for the

polynomials. The reason for all this is simple: what is clear at the development stage and a week after that to the developer, may not be so clear in a month, or to someone other the developer working with the source code.

Careful consideration on variable and function naming conventions is a key not only for future maintenance, but also for debugging. Since the development of the RT solver relies heavily on mathematical apparatus, the developer experiences a natural temptation to use "spelled" symbols from equations. For example, `tau`, `mu`, and `mu0`, for optical thickness τ and cosines of the view μ and solar μ_0 zeniths, respectively. If **`mu`**`[:]` is an array, using of `imu` as an index and `mui=`**`mu`**`[imu]` is convenient. Another example is single scattering albedo, ω_0. What would be the best name: equation based (`omg0` or `omega0`) vs. acronym-based (`ssa`), vs. full or partially spelled (`SingleScatteringAlbedo` – camel case; `sgl_scat_alb` – underscore, etc.)? The answer depends on personal preferences and coding experience, general language standards (e.g., for Python code[26]), overall goal (use the code once and forget vs. create and maintain for years), or naming convention in the team, to name just a few factors. As a rule, we'd recommend using shorter names for frequently used variables. According to this rule and to better represent the structure of the code, the developer might prefer longer names for functions. In Table 5, we list names of all functions we used in the original manuscript and the code before peer-revision, and after it. The intention of the table is to help the developer chose between compact but less meaningful vs. long but descriptive function names.

Table 5: Function names in gsit: compact (originally submitted code) vs expanded (after peer-revision).

Compact	Expanded	Purpose of function
gauszw	gauss_zeroes_weights	Computes Gauss zeros and weights
gsitm	gauss_seidel_iterations_m	Computes m-th Fourier component for multiple scattering of light using Gauss-Seidel iterations
polleg	legendre_polynomial	Computes ordinary Legendre polynomials
polqkm	schmidt_polynomial	Computes Schmidt polynomials for the given zenith moment *k* and azimuth moment *m*
sfidn	source_function_integrate_down	Computes RTE solution at user-defined line of sight at BOA (downward) using integration of the source function
sfiup	source_function_integrate_up	Same as above, except for TOA (upward)
sglsdn	single_scattering_down	Computes single scattering at BOA (downward)
sglsup	single_scattering_up	Same as above, except for TOA (upward)

Object-oriented programming is becoming popular in scientific software development. The object-oriented principles help reduce the number of arguments in functions, and support maintenance as the software is scaled up to include more features. However, the RT solver is a small-scale software containing from a few dozens to a few hundred lines of code. The number of input optical parameters is known in advance from the RT equation, while numerical parameters come from the selected numerical technique. At the low level, the functional form is arguably easier to understand for domain scientists who were taught some programming rather than people who are primarily computer scientists. The object-oriented concept could be advantageous in cases when several numerical RT solvers with

[26] https://www.python.org/dev/peps/pep-0008/

different accuracy parameters on input and maybe different output (flux, intensity only, complete Stokes vector, etc.), are combined in a single package (RT toolbox), which eventually becomes a part of weather forecasting [116] and climate models [117–120].

Finally, one may ask: how long does it take to develop a (simple) RT solver? This, of course, depends on many risk factors [121]. In 1975, Brooks mentioned 2000-3000 debugged instructions (lines) per man-year, which is about 10 lines of code per workday. Decades later in the 1995 edition of the same book, this number did not change [122]. Depending on language, this parameter may reach tens of lines of code per day [123] (not including documentation). In view of the complexity of RT theory we assume 10 lines per day to develop, test, and document the code, and estimate that an RT freshman would spend from two weeks to a month full time to develop and understand the presented code from scratch. The time will not be spent in vain because, as discussed in the Introduction, the useful life spans of RT codes can be years or decades.

5. Summary

In this paper we report, step by step, the process of creating a deterministic (non-Monte Carlo) RT code and relate each step to the underlying theoretical components. As the algorithm of numerical solution to the RTE is essentially opposite to the sequence of sophisticated analytical evaluations (**Fig.17**, red arrows), attempting to code an RT solver based on the existing literature may not offer quick help to an interested user.

For clarity of academic demonstration, our open source[27] Python RT code `gsit` is limited to simulation of monochromatic solar light scattering in a plane-parallel vertically homogeneous atmosphere bounded from below by an evenly reflecting surface. Polarization of light is ignored, optical properties of the atmosphere are assumed known, and only one method (Gauss-Seidel iteration) is discussed in detail. However, we cover the main steps used to solve the RTE in general: expansion of the phase function in Legendre series, Fourier expansion over azimuth for user-defined solar zenith angle, numerical integration over zenith and optical depth, integration of the source function technique to account for user-defined view geometry, and single scattering correction. We show in Section 3.6.1 the difference between the methods of Gauss-Seidel iterations and Successive Orders. We also illustrate the principles of Adding-Doubling and similar techniques. Our Section 4 contains sufficient information on how to make `gsit` applicable for inhomogeneous atmospheres. Pieces of Python code and corresponding necessary equations are collocated and discussed in parallel which, we believe, support understanding.

The analytical integration over optical depth in the methods of Discrete Ordinates and Spherical Harmonics is based on a more sophisticated formalism, not covered in this paper. The Monte Carlo method, which is also relatively easy to understand and code, is also omitted as it has a statistical nature completely different from deterministic RT codes discussed in this paper.

If one wants to start coding quickly, we recommend the following strategy: start from Section 3 by re-typing (not copy-pasting) the presented pieces of code, try to understand relevant equations, and exercise unit testing as described in that section. Numerical implementation of the theoretical evaluations starts with coding the low-level functions first: Legendre polynomials, Gauss nodes and weights, and single scattering approximation. By the end of Section 3, the user will have a fully working

[27] https://github.com/korkins/gsit

RT code (within the limitations mentioned above) which would simplify understanding of Sections 1-2 as well as other literature and existing codes in the Earth science and beyond. Implementation of the inhomogeneous atmosphere in `gsit` would serve as a fairly straightforward exercise, while adding azimuth-dependent BRDF and polarization would require more effort and understanding.

At last, we would like to turn back to Ken Thompson's epigraph at the beginning of the paper. This quotation is perhaps an exaggeration, as there is no doubt that re-use of stable, well-documented, professionally created code is an important part of efficient software development and maintenance. We do not want to dissuade the reader from this, and indeed the present paper imports several community-developed libraries. However, we wish to emphasize that writing (and debugging) a piece of code oneself greatly benefits one's understanding of how it works, and time spent for refactoring of an old-but-good RT code that "works" will pay off in the long run.

Acknowledgements

The work of AI and AMS is supported by the NASA PACE project. The authors also acknowledge the encouragement and support of P. J. Werdell (NASA GSFC) in pursuing this work.

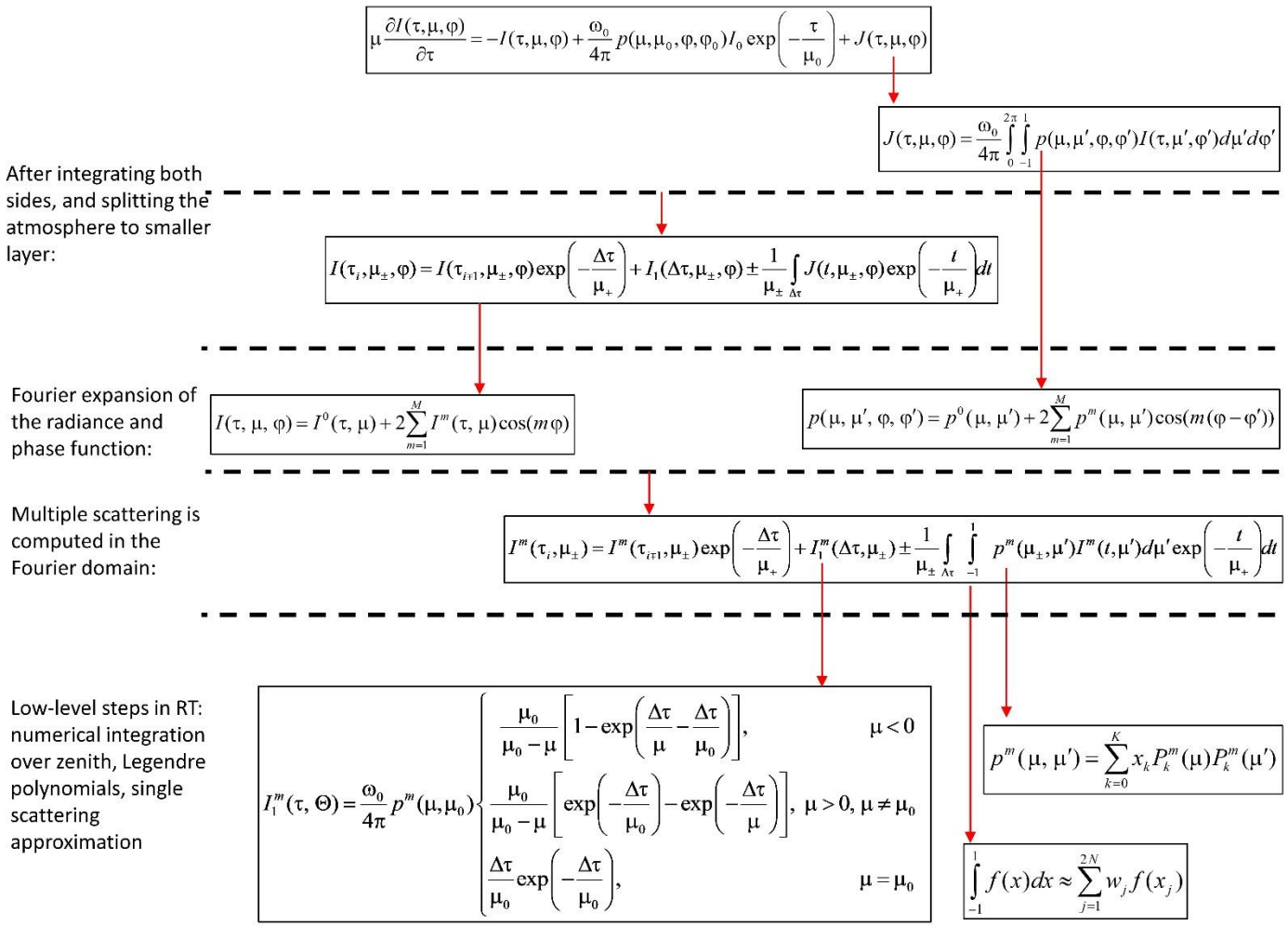


Fig.17: A diagram of the theoretical evaluations starting from the RTE and breaking it down to lower-level steps (red arrows). The development of the RT code, however, starts from low-level stand-alone functions and goes the way opposite to theoretical evaluations.

References

[1] K. Thompson, Commun. ACM 27 (1984) 761–763.

[2] C. Emde, R. Buras-Schnell, A. Kylling, B. Mayer, J. Gasteiger, U. Hamann, J. Kylling, B. Richter, C. Pause, T. Dowling, L. Bugliaro, Geosci. Model Dev. 9 (2016) 1647–1672.

[3] P. Eriksson, S.A. Buehler, C.P. Davis, C. Emde, O. Lemke, J. Quant. Spectrosc. Radiat. Transf. 112 (2011) 1551–1558.

[4] J. Yang, S. Ding, P. Dong, L. Bi, B. Yi, J. Quant. Spectrosc. Radiat. Transf. 251 (2020) 107043.

[5] J.V. Dave, Astrophys. J. 140 (1964) 1292–1303.

[6] P.K. Bhartia, R.D. McPeters, L.E. Flynn, S. Taylor, N.A. Kramarova, S. Frith, B. Fisher, M. Deland, Atmos. Meas. Tech. 6 (2013) 2533–2548.

[7] J. Herman, L. Huang, R. McPeters, J. Ziemke, A. Cede, K. Blank, Atmos. Meas. Tech. 11 (2018) 177–194.

[8] N.A. Krotkov, L.N. Lamsal, E.A. Celarier, W.H. Swartz, S. V. Marchenko, E.J. Bucsela, K.L. Chan, M. Wenig, Atmos. Meas. Tech. Discuss. (2017) 1–42.

[9] L. Lamsal, N. Krotkov, A. Vasilkov, S. Marchenko, W. Qin, E.-S. Yang, Z. Fasnacht, J. Joiner, S. Choi, D. Haffner, W. Swartz, B. Fisher, E. Bucsela, Atmos. Meas. Tech. Discuss. (2020) 1–56.

[10] K. Stamnes, S.-C. Tsay, W. Wiscombe, K. Jayaweera, Appl. Opt. 27 (1988) 2502–2509.

[11] I. Laszlo, K. Stamnes, W. Wiscombe, S.-C. Tsay, in: A.A. Kokhanovsky (Ed.), Light Scatt. Rev., Springer, Berlin, 2016, pp. 3–65.

[12] B. Yao, C. Liu, S. Teng, L. Bi, Z. Zhang, P. Zhang, B.-J. Sohn, Sci. China Earth Sci. 63 (2020) 1701–1713.

[13] D. Ramon, F. Steinmetz, D. Jolivet, M. Compiègne, R. Frouin, J. Quant. Spectrosc. Radiat. Transf. 222–223 (2019) 89–107.

[14] M. Baes, J.I. Davies, H. Dejonghe, S. Sabatini, S. Roberts, R. Evans, S.M. Linder, R.M. Smith, W.J.G. de Blok, Mon. Not. R. Astron. Soc. 343 (2003) 1081–1094.

[15] M. Baes, P. Camps, Astron. Comput. 12 (2015) 33–44.

[16] P. Camps, M. Baes, Astron. Comput. 9 (2015) 20–33.

[17] S. Verstocken, D. Van De Putte, P. Camps, M. Baes, Astron. Comput. 20 (2017) 16–33.

[18] P. Camps, M. Baes, Astron. Comput. 31 (2020) 100381.

[19] Z. Merali, Nature 467 (2010) 775–777.

[20] D. Heaton, J.C. Carver, Inf. Softw. Technol. 67 (2015) 207–219.

[21] O. Dubovik, Z. Li, M.I. Mishchenko, D. Tanré, Y. Karol, B. Bojkov, B. Cairns, D.J. Diner, W.R. Espinosa, P. Goloub, X. Gu, O. Hasekamp, J. Hong, W. Hou, K.D. Knobelspiesse, J. Landgraf, L. Li, P. Litvinov, Y. Liu, A. Lopatin, T. Marbach, H. Maring, V. Martins, Y. Meijer, G. Milinevsky, S. Mukai, F. Parol, Y. Qiao, L. Remer, J. Rietjens, I. Sano, P. Stammes, S. Stamnes, X. Sun, P. Tabary, L.D.

Travis, F. Waquet, F. Xu, C. Yan, D. Yin, J. Quant. Spectrosc. Radiat. Transf. 224 (2019) 474–511.

[22] J. Lenoble, ed., Radiative Transfer in Scattering and Absorbing Atmospheres: Standard Computational Procedures, A Deepak Publishing, Hampton, VA, 1985.

[23] J.E. Hansen, L.D. Travis, Space Sci. Rev. 16 (1974) 527–610.

[24] J.W. Hovenier, C.V.M. der Mee, H. Domke, Transfer of Polarized Light in Planetary Atmospheres: Basic Concepts and Practical Methods, Kluwer Academic Publishers, Dordrecht, 2004.

[25] K. Stamnes, Rev. Geophys. 24 (1986) 299–310.

[26] J. Chowdhary, P.W. Zhai, E. Boss, H. Dierssen, R. Frouin, A. Ibrahim, Z. Lee, L.A. Remer, M. Twardowski, F. Xu, X. Zhang, M. Ottaviani, W.R. Espinosa, D. Ramon, Front. Earth Sci. 7 (2019).

[27] S. Chandrasekhar, Radiative Transfer, Oxford University Press, London, 1950.

[28] V. Sobolev, Light Scattering in Planetary Atmospheres, Pergamon Press, 1976.

[29] V. Kourganoff, Basic Methods in Transfer Problems: Radiative Equlibrium and Neutron Diffusion, Dover Publications, New York, 1952.

[30] K. Stamnes, G.E. Thomas, J.J. Stamnes, Radiative Transfer in the Atmosphere and Ocean, Cambridge University Press, 2017.

[31] V. V Rozanov, A. V Rozanov, A.A. Kokhanovsky, J.P. Burrows, J. Quant. Spectrosc. Radiat. Transf. 133 (2014) 13–71.

[32] R. Spurr, M. Christi, in: A.A. Kokhanovsky (Ed.), Springer Ser. Light Scatt., Springer, 2019, pp. 1–62.

[33] S.Y. Kotchenova, E.F. Vermote, R. Levy, A. Lyapustin, Appl. Opt. 47 (2008) 2215–2226.

[34] J.E. Hansen, J.B. Pollack, J. Atmos. Sci. 27 (1970) 265–281.

[35] H.C. van de Hulst, Multiple Light Scattering: Tables, Formulas, and Applications. Volume 1, Academic Press, 1980.

[36] V. V Rozanov, D. Diebel, R.J.D. Spurr, J.P. Burrows, J. Geophys. Res. Atmos. 102 (1997) 16683–16695.

[37] D.S. Efremenko, D.G. Loyola, A. Doicu, R.J.D. Spurr, Comput. Phys. Commun. 185 (2014) 3079–3089.

[38] S. Korkin, A. Lyapustin, A. Sinyuk, B. Holben, Remote Sens. Clouds Atmos. XXI 10001 (2016) 100010B.

[39] S.V. Korkin, A.I. Lyapustin, V.V. Rozanov, J. Quant. Spectrosc. Radiat. Transf. 112 (2011).

[40] S. Korkin, A. Lyapustin, A. Sinyuk, B. Holben, A. Kokhanovsky, J. Quant. Spectrosc. Radiat. Transf. 200 (2017) 295–310.

[41] S. Korkin, A. Lyapustin, J. Quant. Spectrosc. Radiat. Transf. 227 (2019) 106–110.

[42] A.I. Lyapustin, T.Z. Muldashev, J. Quant. Spectrosc. Radiat. Transf. 61 (1999) 545–555.

[43] A. Lyapustin, Appl. Opt. 41 (2002) 5607.

[44] A.M. Sayer, G.E. Thomas, R.G. Grainger, Atmos. Meas. Tech. 3 (2010) 813–838.

[45] A.M. Sayer, N.C. Hsu, C. Bettenhausen, J. Lee, J. Redemann, B. Schmid, Y. Shinozuka, J. Geophys. Res. 121 (2016) 4830–4854.

[46] A.M. Sayer, N.C. Hsu, J. Lee, C. Bettenhausen, W. V. Kim, A. Smirnov, J. Geophys. Res. Atmos. 123 (2018) 380–400.

[47] A. Ibrahim, A. Gilerson, T. Harmel, A. Tonizzo, J. Chowdhary, S. Ahmed, Opt. Express 20 (2012) 25662.

[48] A. Ibrahim, A. Gilerson, J. Chowdhary, S. Ahmed, Remote Sens. Environ. 186 (2016) 548–566.

[49] A. Ibrahim, B.A. Franz, Z. Ahmad, S.W. Bailey, Front. Earth Sci. 7 (2019) 1–15.

[50] A. Lyapustin, Y. Wang, S. Korkin, D. Huang, Atmos. Meas. Tech. 11 (2018).

[51] A. Lyapustin, Y. Wang, S. Korkin, R. Kahn, D. Winker, IEEE Geosci. Remote Sens. Lett. 17 (2020).

[52] P.J. Werdell, M.J. Behrenfeld, P.S. Bontempi, E. Boss, B. Cairns, G.T. Davis, B.A. Franz, U.B. Gliese, E.T. Gorman, O. Hasekamp, K.D. Knobelspiesse, A. Mannino, J.V. Martins, C. McClain, G. Meister, L.A. Remer, Bull. Am. Meteorol. Soc. 100 (2019) 1775–1794.

[53] P.J. Werdell, B.A. Franz, S.W. Bailey, G.C. Feldman, E. Boss, V.E. Brando, M. Dowell, T. Hirata, S.J. Lavender, Z.P. Lee, H. Loisel, S. Maritorena, F. Mélin, T.S. Moore, T.J. Smyth, D. Antoine, E. Devred, O.H.F. D'Andon, A. Mangin, Appl. Opt. 52 (2013) 2019–2037.

[54] P. Jeremy Werdell, C.R. McClain, Satellite Remote Sensing: Ocean Color, 3rd ed., Elsevier Ltd., 2019.

[55] W.H. Press, B.P. Flannery, S.A. Teukolsky, W.T. Vetterling, Numerical Recipes in Pascal: The Art of Scientific Computing, Cambridge University Press, 1989.

[56] W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes: The Art of Scientific Computing, 3rd ed., Cambridge University Press, 2007.

[57] S. Arms, J. Chastang, M. Grover, J. Thielen, M. Wilson, D. Dirks, Bull. Am. Meteorol. Soc. 101 (2020) E1492–E1496.

[58] R.T. Wilson, Comput. Geosci. 51 (2013) 166–171.

[59] P. Ricchiazzi, S. Yang, C. Gautier, D. Sowle, Bull. Am. Meteorol. Soc. 79 (1998) 2101–2114.

[60] V.P. Ghate, P. Kollias, S. Crewell, A.M. Fridlind, T. Heus, U. Löehnert, M. Maahn, G.M. McFarquhar, D. Moisseev, M. Oue, M. Wendisch, C. Williams, Bull. Am. Meteorol. Soc. 100 (2019) ES5–ES9.

[61] J. Lenoble, M. Herman, J.L. Deuzé, B. Lafrance, R. Santer, D. Tanré, J. Quant. Spectrosc. Radiat. Transf. 107 (2007) 479–507.

[62] O.P. Hasekamp, J. Landgraf, J. Quant. Spectrosc. Radiat. Transf. 75 (2002) 221–238.

[63] A. Lyapustin, Y. Knyazikhin, Appl. Opt. 40 (2001) 3495.

[64] K.F. Evans, J. Atmos. Sci. 55 (1998) 429–446.

[65] A. Doicu, M.I. Mishchenko, D.S. Efremenko, T. Trautmann, J. Quant. Spectrosc. Radiat. Transf. 258 (2021) 107386.

[66] D. Zawada, G. Franssens, R. Loughman, A. Mikkonen, A. Rozanov, C. Emde, A. Bourassa, S. Dueck, H. Lindqvist, D. Ramon, V. Rozanov, E. Dekemper, E. Kyrölä, J.P. Burrows, D. Fussen, D. Degenstein, Atmos. Meas. Tech. 14 (2021) 3953–3972.

[67] P. Castellanos, A. da Silva, J. Atmos. Ocean. Technol. 36 (2019) 819–832.

[68] M. Raissi, P. Perdikaris, G.E. Karniadakis, J. Comput. Phys. 378 (2019) 686–707.

[69] G.E. Karniadakis, I.G. Kevrekidis, L. Lu, P. Perdikaris, S. Wang, L. Yang, Nat. Rev. Phys. 3 (2021) 422–440.

[70] M. Gao, P. Zhai, B.A. Franz, K. Knobelspiesse, A. Ibrahim, B. Cairns, S.E. Craig, G. Fu, O. Hasekamp, Y. Hu, P.J. Werdell, Atmos. Meas. Tech. 13 (2020) 3939–3956.

[71] C. Emde, V. Barlakas, C. Cornet, F. Evans, S. Korkin, Y. Ota, L.C. Labonnote, A. Lyapustin, A. Macke, B. Mayer, M. Wendisch, J. Quant. Spectrosc. Radiat. Transf. 164 (2015) 8–36.

[72] B.M. Herman, S.R. Browning, J. Atmos. Sci. 22 (1965) 559–566.

[73] P.-W. Zhai, Y. Hu, C.R. Trepte, P.L. Lucker, Opt. Express 17 (2009) 2057.

[74] B.M. Herman, T.R. Caudill, D.E. Flittner, K.J. Thome, A. Ben-David, Appl. Opt. 34 (1995) 4563–4572.

[75] A.H. Karp, J. Greenstadt, J.A. Fillmore, J. Quant. Spectrosc. Radiat. Transf. 24 (1980) 391–406.

[76] A.I. Lyapustin, Appl. Opt. 44 (2005) 7764–7772.

[77] C. Moler, C. Van Loan, SIAM Rev. 45 (2003) 3–49.

[78] S. Korkin, E.-S. Yang, R. Spurr, C. Emde, N. Krotkov, A. Vasilkov, D. Haffner, J. Mok, A. Lyapustin, J. Quant. Spectrosc. Radiat. Transf. 254 (2020) 107181.

[79] V.I. Krylov, A.H. Stroud, Approximate Calculation of Integrals, Dover, 2005.

[80] W.H. Press, S. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes in Fortran 77: The Art of Scientific Computing, 2nd ed., 1997.

[81] W.H. Press, S. Teukolsky, W.T. Vetterling, B.P. Flannery, Numerical Recipes in C: The Art of Scientific Computing, 2nd ed., Cambridge University Press, 1997.

[82] J.B. Sykes, Mon. Not. R. Astron. Soc. 111 (1951) 377–386.

[83] H.J. Weber, G.B. Arfken, Essential Mathematical Methods for Physicists, 2003.

[84] R. Blackely, Potential Theory in Gravity and Magnetic Applications, University Press, Cambridge, 1995.

[85] W.M.F. Wauben, J.F. de Haan, J.W. Hovenier, Astron. Astrophys. 276 (1993) 589.

[86] B.M. Herman, J. Geophys. Res. 70 (1965) 1215–1225.

[87] B. Van Diedenhoven, O.P. Hasekamp, J. Landgraf, Appl. Opt. 45 (2006) 5993–6006.

[88] B.M. Herman, A. Ben-David, K.J. Thome, Appl. Opt. 33 (1994) 1760.

[89] T.R. Caudill, D.E. Flittner, B.M. Herman, O. Torres, R.D. McPeters, J. Geophys. Res. Atmos. 102 (1997) 3881–3890.

[90] R. Loughman, D. Flittner, E. Nyaku, P.K. Bhartia, Atmos. Chem. Phys. 15 (2015) 3007–3020.

[91] J. Eluszkiewicz, G. Uymin, D. Flittner, K. Cady-Pereira, E. Mlawer, J. Henderson, J.L. Moncet, T. Nehrkorn, M. Wolff, J. Quant. Spectrosc. Radiat. Transf. 193 (2017) 31–39.

[92] H.H. Walter, J. Landgraf, O.P. Hasekamp, J. Quant. Spectrosc. Radiat. Transf. 85 (2004) 251–283.

[93] H.H. Walter, J. Landgraf, J. Quant. Spectrosc. Radiat. Transf. 95 (2005) 175–200.

[94] Z. Ahmad, R.S. Fraser, J. Atmos. Sci. 39 (1982) 656–665.

[95] J. V Dave, J. Gazdag, Appl. Opt. 9 (1970) 1457–1466.

[96] M.I. Mishchenko, A.A. Lacis, L.D. Travis, J. Quant. Spectrosc. Radiat. Transf. 51 (1994) 491–510.

[97] E.S. Chalhoub, R.D.M. Garcia, J. Quant. Spectrosc. Radiat. Transf. 64 (2000) 517–535.

[98] P.W. Zhai, Y. Hu, D.B. Josset, C.R. Trepte, P.L. Lucker, B. Lin, J. Quant. Spectrosc. Radiat. Transf. 115 (2013) 19–27.

[99] F. Kuik, J.F. de Haan, J.W. Hovenier, J. Quant. Spectrosc. Radiat. Transf. 47 (1992) 477–489.

[100] C.F. Bohren, D.R. Huffman, Absorption and Scattering of Light by Small Particles, Wiley-VCH, 1998.

[101] W.J. Wiscombe, Appl. Opt. 19 (1980) 1505–1509.

[102] M.I. Mishchenko, L.D. Travis, A.A. Lacis, Scattering, Absorption, and Emission of Light by Small Particles, Cambridge University Press, Cambridge, 2002.

[103] M.I. Mishchenko, J. Quant. Spectrosc. Radiat. Transf. 242 (2020) 106692.

[104] O. Dubovik, A. Sinyuk, T. Lapyonok, B.N. Holben, M. Mishchenko, P. Yang, T.F. Eck, H. Volten, O. Muñoz, B. Veihelmann, W.J. van der Zande, J.F. Leon, M. Sorokin, I. Slutsker, J. Geophys. Res. Atmos. 111 (2006) 1–34.

[105] S. Oliveira, D.E. Stewart, Writing Scientific Software: A Guide for Good Style, Cambridge University Press, Cambridge, UK, 2006.

[106] R. Gerber, A. Bik, K. Smith, X. Tian, The Software Optimization Cookbook, 2nd ed., Intel Press, 2005.

[107] S. Chellappa, F. Franchetti, M. Puschel, in: R. Lammel, J. Visser, J. Saraiva (Eds.), Gener. Transform. Tech. Softw. Eng. II, Springer, Berlin, 2007, pp. 196–259.

[108] D. Kelly, D. Hook, R. Sanders, Comput. Sci. Eng. 11 (2009) 48–52.

[109] G. Wilson, D.A. Aruliah, C.T. Brown, N.P. Chue Hong, M. Davis, R.T. Guy, S.H.D. Haddock, K.D. Huff, I.M. Mitchell, M.D. Plumbley, B. Waugh, E.P. White, P. Wilson, PLoS Biol. 12 (2014).

[110] J. Carver, D. Heaton, L. Hochstein, R. Bartlett, Comput. Sci. Eng. 15 (2013) 7–11.

[111] B.W. Kernighan, P.J. Plauger, The Elements of Programming Style, 2nd ed., McGraw-Hill, 1978.

[112] K. Dowd, C. Severance, High Performance Computing, 2nd ed., O'Reilly Media, 1998.

[113] G. Hager, G. Wellein, Introduction to High Performance Computing for Scientists and Engineers, 1st ed., CRC Press, Inc., USA, 2010.

[114] R. Buras, T. Dowling, C. Emde, J. Quant. Spectrosc. Radiat. Transf. 112 (2011) 2028–2034.

[115] V. V. Rozanov, A.I. Lyapustin, J. Quant. Spectrosc. Radiat. Transf. 111 (2010) 1964–1979.

[116] R. Kendall, J.C. Carver, D. Fisher, D. Henderson, A. Mark, D. Post, C.E. Rhoades, S. Squires, IEEE Softw. 25 (2008) 59–65.

[117] J.B. Drake, P.W. Jones, G.R. Carr, Int. J. High Perform. Comput. Appl. 19 (2005) 177–186.

[118] S.M. Easterbrook, T.C. Johns, Comput. Sci. Eng. 11 (2009) 65–74.

[119] T. Clune, R. Rood, IEEE Softw. 28 (2011) 49–55.

[120] J. Pipitone, S. Easterbrook, Geosci. Model Dev. 5 (2012) 1009–1022.

[121] R. Sanders, D. Kelly, IEEE Softw. 25 (2008) 21–28.

[122] F.P. Brooks, The Mythical Man-Month (Anniversary Ed.), Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[123] L. Prechelt, Computer (Long. Beach. Calif). 33 (2000) 23–29.

Appendix: Benchmark Results for Section 3.6.3: Main Program

Table 4a. TOA results. Columns left to right: solar zenith angle, SZA, relative azimuth, AZA, cosine of the view zenith angle (negative for ascending radiation), "exact" results generated with IPOL for Rayleigh scattering, "exact" results generated with IPOL for Aerosol scattering over reflecting surface. One can reproduce these results using our open-source Python script gsit.py.

See Table4a_TOA.xls

Table 4b. Same as Table 4a, except for BOA radiation (positive view cosines).

See Table4b_BOA.xls